The 54321 Development Document

Patrick Stein

2001-11-16

Table of Contents

Contents

1 Introduction

This document is geared toward the programmer. It is geared toward the programmer who wishes to understand the way this game is coded. It is not really geared toward the end-user. All of the end-user explanation of "Why Four Dimensions?" and "What's The Object Of This Game?" is dealt with in the "Help" portions of the game.

In general, I tried to follow a "Model-View-Controller" paradigm. I munged this slightly by including some fielding of mouse events in the view instead of in the controller. But, when I have done that, I have had the view invoke methods on the controller to let the controller update the model.

I have tried to adhere to this paradigm so that at a later date, I can replace the controllers with different ones to read in moves from a CGI script in order to verify high-score claims by clients. It is becoming less and less clear to me, how useful this will prove.

The only other bit of munging is that I have made sound a part of the "View". I would argue that this isn't really munging. The "View" is meant to

be the user-presentable version of the model. It just so happens that some of the rendering is aural instead of visual.

This program was written for the `1MB SDL Game Programming` contest sponsored by *Linux Journal*[1], *Loki Software*[2], and *No Starch Press*[3].

To keep the global namespace squeaky clean, almost all of the code herein is wrapped in its own namespace.

⟨*NameSpace*⟩≡
```
NKlein_54321
```

[1] `http://www.linuxjournal.com/`
[2] `http://www.lokigames.com/`
[3] `http://www.nostarch.com/`

Part I
Underlying Basics

2 Generic Cube

The namespace inside the cube class is a concatenation of the general namespace and the name of the cube class.

⟨*CubeNameSpace*⟩≡
  ⟨*NameSpace*⟩`::Cube`

All of the games here are based upon a cube that is length four on each side. This class encapsulates a simple cube (in either two, three, or four dimensions) that is length four on each side. This cube class simply stores an integer for each unit cell and allows access to it.

⟨*Cube Type Declarations*⟩≡
```
typedef int CellType;
```

⟨*Cube Constant Declarations*⟩≡
```
enum { SIDE_LENGTH = 4 };
```

The memory and computational requirements for this class could be significantly diminished for the two- and three-dimensional cases. However, because this code is geared toward a contest where the compiled size of the code matters (and because none of the games require stunning response times), this class doesn't make any allowance for the number of dimensions, it always assumes the four-dimensional case.

⟨*Cube Constant Declarations*⟩+≡
```
enum { DIMENSIONS = 4 };
```

The primary aim of this class is to store an integer at each point in a $4 \times 4 \times 4 \times 4$ cube.

⟨*Cube Constant Declarations*⟩+≡
```
enum {
    ARRAY_LEN = SIDE_LENGTH * SIDE_LENGTH * SIDE_LENGTH * SIDE_LENGTH
};
```

⟨*Cube Internal Array Declaration*⟩≡
```
CellType array[ ARRAY_LEN ];
```

In addition to the whole four-dimensional array length, the class stores a lookup table for the effective array lengths in different dimensions.

⟨*Cube Array Length Declaration*⟩≡
```
static const unsigned int arrayLengths[];
```

The effective `ARRAY_LEN` for a particular number of dimensions is the `SIDE_LEN` raised to the dimensions power.

⟨*Cube Array Length*⟩≡
```
const unsigned int ⟨CubeNameSpace⟩::arrayLengths[] = {
    1,
    SIDE_LENGTH,
    SIDE_LENGTH * SIDE_LENGTH,
    SIDE_LENGTH * SIDE_LENGTH * SIDE_LENGTH,
    SIDE_LENGTH * SIDE_LENGTH * SIDE_LENGTH * SIDE_LENGTH
};
```

2.1 The Constructor

The default constructor is the only one needed. It, of course, requires no arguments.

⟨*Cube Constructor Declaration*⟩≡
```
Cube( void );
```

The constructor for the cube simply initializes each element of the array to zero using the assignment method defined in the following section. The constructor also validates the assumption of all of this code that there are to be (at most) four spatial dimensions.

⟨*Cube Constructor Implementation*⟩≡
```
⟨CubeNameSpace⟩::Cube( void )
{
    assert( DIMENSIONS == 4 );
    CellType zero = (CellType)0;
    *this = zero;
}
```

2.2 Assigning the Whole Cube a Single Value

We will override the assignment operator to allow one to set the whole cube to a single value.

⟨*Cube Assignment Declaration*⟩≡
```
void operator = ( const CellType& value );
```

This is as simple as looping through each possible index in the internal array and assigning the value to each one. But, first, we verify that the value is in a legitimate portion of memory.

⟨*Cube Assignment Implementation*⟩≡
```
void
⟨CubeNameSpace⟩::operator = (
        const ⟨CubeNameSpace⟩::CellType& value
    )
{
    ⟨Cube Check Value Address⟩
    for ( unsigned int ii=0; ii < ARRAY_LEN; ++ii ) {
        this->array[ ii ] = value;
    }
}
```

The only check performed to ensure that the value is valid is to make sure that its address is non-null.

⟨*Cube Check Value Address*⟩≡
```
assert( &value != 0 );
```

2.3 Coordinate Manipulation Methods

In order to facilitate simpler storage in classes that need to maintain references to particular cells, the `Cube` class makes available methods which convert between a coordinate representation and the cube's internal one-dimensional indexing and a method which can take two sets of coordinates that differ on one axis and tell which axis and which direction they differ.

The first of these methods allows one to convert from vector coordinates into the cube's internal indexing.

⟨*Cube Vector To Index Declaration*⟩≡
```
static void vectorToIndex(
        const unsigned int vec[ DIMENSIONS ],
        unsigned int* index
    );
```

This method first verifies that the output location and the input vector are legitmate. Then, it calculates the index represented by the vector and stores it in the output location.

⟨*Cube Vector To Index Implementation*⟩≡

```
void
⟨CubeNameSpace⟩::vectorToIndex(
        const unsigned int vec[ ⟨CubeNameSpace⟩::DIMENSIONS ],
        unsigned int* index
    )
{
    ⟨Cube Check Index Pointer⟩
    ⟨Cube Check Vector⟩
    ⟨Cube Calculate Index⟩
}
```

The only check performed to ensure that the index pointer is valid is to make sure it is non-null.

⟨*Cube Check Index Pointer*⟩≡

```
assert( index != 0 );
```

Verifying the the coordinates are legitimate is as simple as making sure that each coordinate is less than `SIDE_LENGTH`. They are guaranteed to be greater than or equal to zero because they are unsigned integers.

⟨*Cube Check Vector*⟩≡

```
assert( vec[ 0 ] < SIDE_LENGTH );
assert( vec[ 1 ] < SIDE_LENGTH );
assert( vec[ 2 ] < SIDE_LENGTH );
assert( vec[ 3 ] < SIDE_LENGTH );
```

There are a variety of ways we could have calculated the index from the vector. This is the most readable way, in my opinion. If you've done any image processing, you are probably familiar with calculations like `xx + yy * width`. The following calculation is simply an n-dimensional version of that. You can verify this to yourself by assuming that only `vec[0]` and `vec[1]` are non-zero.

⟨*Cube Calculate Index*⟩≡

```
*index = 0;

for ( unsigned int ii=0; ii < DIMENSIONS; ++ii ) {
    *index *= SIDE_LENGTH;
    *index += vec[ ( DIMENSIONS - 1 ) - ii ];
}
```

The second of these methods allows one to convert from the cube's internal indexing into vector coordinates.

⟨*Cube Index To Vector Declaration*⟩≡
```
static void indexToVector(
        unsigned int index,
        unsigned int vec[ DIMENSIONS ]
    );
```

This method first verifies that the index is in the valid range and that the output vector is legitimate. Then, it calculates the vector represented by the given index.

⟨*Cube Index To Vector Implementation*⟩≡
```
void
⟨CubeNameSpace⟩::indexToVector(
        unsigned int index,
        unsigned int vec[ ⟨CubeNameSpace⟩::DIMENSIONS ]
    )
{
    ⟨Cube Check Index Value⟩
    ⟨Cube Check Vector Pointer⟩
    ⟨Cube Calculate Vector⟩
}
```

The check to verify that the index is on a valid range is quite simple. It simply checks to see that the index is smaller than the length of the internal array. Since the index is an unsigned integer, we are already assured that it is at least zero.

⟨*Cube Check Index Value*⟩≡
```
assert( index < ARRAY_LEN );
```

The check to verify that the vector pointer is valid simply checks to be sure that it is non-null. It's an unsophisticated check, but should never really get invoked anyway.

⟨*Cube Check Vector Pointer*⟩≡
```
assert( vec != 0 );
```

This calculation is the inverse of the calculation from the previous method (hopefully, that is no surprise). The above method repeatedly added in coordinates and multiplied by `SIDE_LENGTH`. This method `mod`s out the coordinates and divides by `SIDE_LENGTH`.

⟨*Cube Calculate Vector*⟩≡
```
for ( unsigned int ii=0; ii < DIMENSIONS; ++ii ) {
    vec[ ii ] = index % SIDE_LENGTH;
    index /= SIDE_LENGTH;
}
```

Several classes need to determine the direction of motion between two vectors that are in line with each other. This method returns `true` if the two vectors are on the same line and it assigns the `axis` parameter to the axis on which they differ and the `positive` parameter to reflect whether the distance is shorter in the positive or negative direction.

⟨*Cube Determine Axis Declaration*⟩≡

```
static bool determineAxis(
        unsigned int vf[ DIMENSIONS ],
        unsigned int vt[ DIMENSIONS ],
        bool wrapping,
        unsigned int* axis,
        bool* positive
    );
```

This method counts the number of axises that differ. There should be exactly one differing axis for any valid move.

⟨*Cube Determine Axis Implementation*⟩≡

```
bool
⟨CubeNameSpace⟩::determineAxis(
        unsigned int vf[ DIMENSIONS ],
        unsigned int vt[ DIMENSIONS ],
        bool wrapping,
        unsigned int* axis,
        bool* positive
    )
{
    assert( axis != 0 );
    assert( positive != 0 );

    *positive = true;

    unsigned int diffCount = 0;
    ⟨Cube Calculate Which Axis⟩
    return ( diffCount == 1 );
}
```

To determine which axis the change is on, we take the difference of each coordinate. When we find one that is different, we record which axis it was on and whether the difference was positive or negative.

⟨*Cube Calculate Which Axis*⟩≡

```
for ( unsigned int ii=0; ii < DIMENSIONS; ++ii ) {
    int diff = ( (int)vt[ ii ] - (int)vf[ ii ] );

    if ( diff != 0 ) {
        if ( wrapping ) {
            unsigned int udiff = ( SIDE_LENGTH + diff ) % SIDE_LENGTH;
            if ( udiff >= SIDE_LENGTH/2 ) {
                *positive = false;
            }
        } else {
            if ( diff < 0 ) {
                *positive = false;
            }
        }
        *axis = ii;
        ++diffCount;
    }
}
```

2.4 Access Methods

All of the storage space in the world is pointless if one cannot access it. The `Cube` class allows one to access the elements in the cube by giving their vector coordinates.

⟨*Cube Get Cell Declarations*⟩≡

```
CellType& operator [] ( const unsigned int vec[ DIMENSIONS ] );
```

To do this, the method simply calls the method which converts the vector coordinates into an index and then uses that index to return the appropriate element.

⟨*Cube Get Cell Implementations*⟩≡

```
⟨CubeNameSpace⟩::CellType&
⟨CubeNameSpace⟩::operator [] (
        const unsigned int vec[ ⟨CubeNameSpace⟩::DIMENSIONS ]
    )
{
    unsigned int index;
    this->vectorToIndex( vec, &index );

    return this->array[ index ];
}
```

In addition to allowing one to access the element via vector coordinates, the `Cube` class also allows one to access it by the internal index.

⟨*Cube Get Cell Declarations*⟩+≡
```
CellType& operator [] ( const unsigned int index );
```

To do this, the method simply returns the appropriate element.

⟨*Cube Get Cell Implementations*⟩+≡
```
⟨CubeNameSpace⟩::CellType&
⟨CubeNameSpace⟩::operator [] (
        const unsigned int index
    )
{
    assert( index < ARRAY_LEN );
    return this->array[ index ];
}
```

The cube also has a topology to it. Rather than encode that topology into every one of the subgames, the `Cube` class will have a method which retrieves the indexes of the neighbors of a particular cell. The method returns the number of neighbors found.

⟨*Cube Get Neighbors Declaration*⟩≡
```
static unsigned int getNeighbors(
        unsigned int nn[ 2 * DIMENSIONS ],
        unsigned int index,
        unsigned int dimensions,
        bool wrap = true
    );
```

This method simply goes through each of the possible directions and checks the neighboring coordinates. This method loops through the dimensions of interest. At each one, it checks the element at `+1` and `-1` in that direction.

⟨*Cube Get Neighbors Implementation*⟩≡

```
unsigned int
⟨CubeNameSpace⟩::getNeighbors(
        unsigned int nn[
                2 * ⟨CubeNameSpace⟩::DIMENSIONS
            ],
        unsigned int index,
        unsigned int dims,
        bool wrap
    )
{
    unsigned int vec[ DIMENSIONS ];
    ⟨CubeNameSpace⟩::indexToVector( index, vec );

    unsigned int ii=0;

    if ( wrap ) {
        for ( unsigned int jj=0; jj < dims; ++jj ) {
            ⟨Cube getNeighbors check wrapped⟩
        }
    } else {
        for ( unsigned int jj=0; jj < dims; ++jj ) {
            ⟨Cube getNeighbors check no wrap⟩
        }
    }

    return ii;
}
```

When checking for wrapped coordinates, we simply take the original coordinate and check it with plus or minus one added to it. We take special care when subtracting to ensure that we stay positive the whole time. Then, we restore the original coordinate.

⟨*Cube getNeighbors check wrapped*⟩≡

```
unsigned int coord = vec[ jj ];
vec[ jj ] = ( coord + 1 ) % SIDE_LENGTH;
⟨CubeNameSpace⟩::vectorToIndex( vec, &nn[ ii++ ] );
vec[ jj ] = ( coord + SIDE_LENGTH - 1 ) % SIDE_LENGTH;
⟨CubeNameSpace⟩::vectorToIndex( vec, &nn[ ii++ ] );
vec[ jj ] = coord;
```

When checking in the non-wrapped case, we have to make sure not to include elements that run off the edges.

⟨*Cube getNeighbors check no wrap*⟩≡
```
unsigned int coord = vec[ jj ];
if ( coord + 1 < SIDE_LENGTH ) {
    vec[ jj ] = ( coord + 1 );
    ⟨CubeNameSpace⟩::vectorToIndex( vec, &nn[ ii++ ] );
}
if ( coord >= 1 ) {
    vec[ jj ] = ( coord - 1 );
    ⟨CubeNameSpace⟩::vectorToIndex( vec, &nn[ ii++ ] );
}
vec[ jj ] = coord;
```

2.5 The `Cube` class

In this section, we assemble the `Cube` class from the pieces in the sections above.

First, we include the type and constant declarations so that they will be readily available for use in other declarations.

⟨*Cube Class Definition*⟩≡
```
public:
    ⟨Cube Type Declarations⟩
    ⟨Cube Constant Declarations⟩
```

Then, we declare the constructor for the cube class.

⟨*Cube Class Definition*⟩+≡
```
public:
    ⟨Cube Constructor Declaration⟩
```

After the constructor, we declare the method used to assign the same value to each element of the cube.

⟨*Cube Class Definition*⟩+≡
```
public:
    ⟨Cube Assignment Declaration⟩
```

The next methods that we include are the vector to index and index to vector methods and the axis determination routine.

⟨*Cube Class Definition*⟩+≡
```
public:
    ⟨Cube Index To Vector Declaration⟩
    ⟨Cube Vector To Index Declaration⟩
    ⟨Cube Determine Axis Declaration⟩
```

After that, we declare the accessor method for the cells of this class and the accessor method that retrieves neighbors of a given point.

⟨*Cube Class Definition*⟩+≡
```
public:
    ⟨Cube Get Cell Declarations⟩
    ⟨Cube Get Neighbors Declaration⟩
```

Next, we declare the internal storage portion of the class.

⟨*Cube Class Definition*⟩+≡
```
private:
    ⟨Cube Internal Array Declaration⟩
```

We also declare the lookup table which stores the effective array length for different dimensions.

⟨*Cube Class Definition*⟩+≡
```
public:
    ⟨Cube Array Length Declaration⟩
```

Once these declarations are all done, we throw all of these together into the class declaration itself.

⟨*Cube Class Declaration*⟩≡
```
class Cube {
    ⟨Cube Class Definition⟩
};
```

2.6 The `cube.h` file

In this section, we assemble the header file for the `Cube` class. It is really straightforward since we assembled the class declaration in the previous section. The only thing that we add to the class declaration is that we tuck it into our own name space so that we can keep the global namespace squeaky clean.

⟨*cube.h*⟩≡
```
namespace ⟨NameSpace⟩ {
    ⟨Cube Class Declaration⟩
};
```

2.7 The `cube.cpp` file

For the actual C++ source code, we include the header file that defines `assert()` and then include the header file generated in the previous section.

⟨*cube.cpp*⟩≡
```
#include <assert.h>
#include "cube.h"
```

Next, we include the actual lookup table for the effective array lengths for different dimensions.

⟨*cube.cpp*⟩+≡
```
⟨Cube Array Length⟩
```

After that, we incorporate the implementation of the constructor.

⟨*cube.cpp*⟩+≡
⟨*Cube Constructor Implementation*⟩

The source file also includes the code for the method which assigns each element of the cube a given value.

⟨*cube.cpp*⟩+≡
⟨*Cube Assignment Implementation*⟩

Next, we include the implementations of the vector to index and index to vector methods and the axis determination routine.

⟨*cube.cpp*⟩+≡
⟨*Cube Vector To Index Implementation*⟩
⟨*Cube Index To Vector Implementation*⟩
⟨*Cube Determine Axis Implementation*⟩

And, finally, we incorporate the implementation of the accessor method for the cells of the cube and the method which retrieves neighbors of a given cell.

⟨*cube.cpp*⟩+≡
⟨*Cube Get Cell Implementations*⟩
⟨*Cube Get Neighbors Implementation*⟩

3 The Font Class

The namespace inside the font class is a concatenation of the general namespace and the name of the Font class.

⟨*FontNameSpace*⟩≡
⟨*NameSpace*⟩`::Font`

The font class stores a pointer to the image used for the font.

⟨*Font Image*⟩≡
```
SDL_Surface* image;
```

The font class also keeps track of the width of each character in the font.

⟨*Font Widths*⟩≡
```
enum {
    START_CHAR = 32,
    END_CHAR = 127
};
unsigned int widths[ END_CHAR+1 ];
```

3.1 The Constructor

The constructor for the font class takes no arguments.

⟨*Font Constructor Declaration*⟩≡
```
Font( void );
```

The constructor for the font class simply loads the font image and fills in the character widths. Each character in the font begins one pixels from the left edge and sixteen pixels from the top edge. The widths of each character are different.

⟨*Font Constructor Implementation*⟩≡
```
⟨FontNameSpace⟩::Font( void )
{
    this->image = ::IMG_Load( "../../data/font.png" );
    for ( unsigned int ii=0; ii < START_CHAR; ++ii ) {
        this->widths[ ii ] = 0;
    }
    for ( unsigned int ii=START_CHAR; ii < END_CHAR; ++ii ) {
        this->widths[ ii ] = 8;
    }

    ⟨Font prepare character widths⟩
}
```

There are plenty of better ways to assign each of the characters a width. But, this one seemed the simplest to me at the time. Since the default width of all characters is 8, those characters that are to stay that width are not included. Those characters which are included are included in the order they appear in the font bitmap (top-to-bottom, left-to-right).

⟨*Font prepare character widths*⟩≡

```
this->widths[ ' ' ] = 4;
this->widths[ ',' ] = 4;
this->widths[ 't' ] = 6;

this->widths[ '!' ] = 5;
this->widths[ '-' ] = 7;
this->widths[ 'E' ] = 7;
this->widths[ ']' ] = 6;
this->widths[ 'i' ] = 4;

this->widths[ '"' ] = 7;
this->widths[ '.' ] = 4;
this->widths[ ':' ] = 4;
this->widths[ 'F' ] = 7;
this->widths[ 'j' ] = 6;

this->widths[ '#' ] = 12;
this->widths[ ';' ] = 4;
this->widths[ 'G' ] = 7;
this->widths[ '_' ] = 12;
this->widths[ 'w' ] = 13;

this->widths[ '$' ] = 7;
this->widths[ '<' ] = 10;
this->widths[ 'T' ] = 7;
this->widths[ '`' ] = 5;
this->widths[ 'l' ] = 4;
this->widths[ 'x' ] = 9;

this->widths[ '%' ] = 11;
this->widths[ '1' ] = 5;
this->widths[ 'I' ] = 5;
this->widths[ 'a' ] = 7;
this->widths[ 'm' ] = 11;

this->widths[ '>' ] = 10;
this->widths[ 'V' ] = 9;
this->widths[ 'n' ] = 7;
```

```
this->widths[ '\'' ] = 3;
this->widths[ 'W' ] = 14;
this->widths[ 'c' ] = 7;
this->widths[ '{' ] = 6;

this->widths[ '(' ] = 5;
this->widths[ '@' ] = 16;
this->widths[ 'L' ] = 7;
this->widths[ 'X' ] = 9;
this->widths[ '|' ] = 4;

this->widths[ ')' ] = 5;
this->widths[ '5' ] = 7;
this->widths[ 'A' ] = 7;
this->widths[ 'M' ] = 12;
this->widths[ 'q' ] = 7;
this->widths[ '}' ] = 6;

this->widths[ '*' ] = 7;
this->widths[ '6' ] = 7;
this->widths[ 'B' ] = 7;
this->widths[ 'Z' ] = 6;
this->widths[ 'f' ] = 5;
this->widths[ 'r' ] = 7;

this->widths[ '7' ] = 7;
this->widths[ 'C' ] = 7;
this->widths[ 'O' ] = 7;
this->widths[ '[' ] = 4;
this->widths[ 'g' ] = 7;
this->widths[ 's' ] = 7;
```

3.2 The Destructor

The destructor for the font class releases the image loaded in the constructor.

⟨*Font Destructor Declaration*⟩≡

```
~Font( void );
```

The destructor for the font class simply releases the font image.

⟨*Font Destructor Implementation*⟩≡

```
⟨FontNameSpace⟩::~Font( void )
{
    if ( this->image != 0 ) {
        ::SDL_FreeSurface( this->image );
    }
}
```

3.3 The Display Methods

The font class has a method which allows one to write text centered at a specific point on a surface.

⟨*Font Display Declarations*⟩≡
```
virtual void centerMessage(
        SDL_Surface* screen,
        bool refresh,
        int xx, int yy,
        const char* fmt,
        ...
    );
```

The display function here uses `vsnprintf()` function to format the message. Then, it calculates the width of the message. Then, it calculates the point it should start displaying the message and displays it.

⟨*Font Display Implementations*⟩≡
```
void
⟨FontNameSpace⟩::centerMessage(
        SDL_Surface* screen,
        bool refresh,
        int xx, int yy,
        const char* fmt,
        ...
    )
{
    ⟨Font centerMessage prepare string⟩
    ⟨Font centerMessage calculate width⟩
    ⟨Font centerMessage calculate starting point⟩
    ⟨Font centerMessage display text⟩
}
```

If you've ever used `vsnprintf()` or its kin before, you should recognize this. Here, we assume the a buffer of size 256 will be big enough for any message that would actually fit on the screen.

⟨*Font centerMessage prepare string*⟩≡
```
char buf[ 256 ];
va_list ap;
va_start( ap, fmt );
vsnprintf( buf, sizeof(buf), fmt, ap );
va_end( ap );
```

To calculate the width, we simply start summing the widths of each character in the string.

⟨*Font centerMessage calculate width*⟩≡

```
unsigned int width = 0;
for ( const char* ptr = buf; *ptr != 0; ++ptr ) {
    assert( (unsigned int)*ptr <= END_CHAR );
    width += this->widths[ *ptr ];
}
```

The starting point should be half the width to the left of the base point and it should be sixteen pixels above the baseline. The extra `1` subtracted from the `xx` coordinate is to compensate for the fact that the first column of each character is not to be considered in the placing of the character.

⟨*Font centerMessage calculate starting point*⟩≡

```
xx -= (int)width / 2 + 1;
yy -= 16;
```

The message is displayed by blitting each character to the screen. Once the whole message has been blitted, the whole area will be updated. The extra `32` added to the width to refresh allows for the fact that we started one pixel to the left of the origin and the last letter may extend further to the right than its width would specify.

⟨*Font centerMessage display text*⟩≡

```
SDL_Rect dst;
dst.x = xx;
dst.y = yy;

for ( const char* ptr = buf; *ptr != 0; ++ptr ) {
    if ( *ptr >= START_CHAR ) {
        unsigned int cc = *ptr - START_CHAR;
        unsigned int row = cc / 12;
        unsigned int col = cc - row * 12;
        SDL_Rect src;
        src.x = col * 21;
        src.y = row * 32;
        src.w = this->widths[ *ptr ];
        src.h = 32;
        ::SDL_BlitSurface( this->image, &src, screen, &dst );
        dst.x += src.w;
    }
}

if ( refresh ) {
    ::SDL_UpdateRect( screen, xx, yy, width+32, 32 );
}
```

3.4 The `Font` class

In this section, we assemble the `Font` class from the pieces in the sections above.

We include, in the `Font` class, the constructor and the display methods.

⟨*Font Class Definition*⟩≡
```
public:
    ⟨Font Constructor Declaration⟩
    ⟨Font Destructor Declaration⟩
    ⟨Font Display Declarations⟩
```

We include the variables that are used in the font class.

⟨*Font Class Definition*⟩+≡
```
private:
    ⟨Font Image⟩
    ⟨Font Widths⟩
```

Once these declarations are all done, we throw all of these together into the class declaration itself.

⟨*Font Class Declaration*⟩≡
```
class Font {
    ⟨Font Class Definition⟩
};
```

3.5 The `font.h` file

In this section, we assemble the header file for the `Font` class. It is really straightforward since we assembled the class declaration in the previous section. The only thing that we add to the class declaration is that we tuck it into our own name space so that we can keep the global namespace squeaky clean.

⟨*font.h*⟩≡
```
namespace ⟨NameSpace⟩ {
    ⟨Font Class Declaration⟩
};
```

3.6 The `font.cpp` file

In this section, we assemble the Font source file. It requires the `SDL` headers for dealing with surfaces, the screen, blitting, and loading images.

⟨*font.cpp*⟩≡
```
#include <SDL.h>
#include <SDL_image.h>
#include <assert.h>
#include <stdarg.h>
#include "font.h"
```

After the header files, we include the implementations of the constructor and the display methods.

⟨*font.cpp*⟩+≡
⟨*Font Constructor Implementation*⟩
⟨*Font Destructor Implementation*⟩
⟨*Font Display Implementations*⟩

4 Sound Device

The namespace inside the sound device class is a concatenation of the general namespace and the name of the sound device class.

⟨*SoundDevNameSpace*⟩≡
  ⟨*NameSpace*⟩`::SoundDev`

The sound device uses a constant to define the frequency that it will open the system audio device.

⟨*SoundDev Frequency*⟩≡
```
enum {
    FREQ = 8192
};
```

The sound device tracks the actual parameters with which the sound device was opened.

⟨*SoundDev Audio Spec*⟩≡
```
SDL_AudioSpec spec;
bool opened;
```

The sound device also keeps track of the size of the buffer that it is currently playing.

⟨*SoundDev Current Buffer*⟩≡
```
Uint8* currentBuf;
Uint8* currentPtr;
unsigned int currentLen;
```

The sound device has a buffer which it keeps set up with a ding in it.

⟨*SoundDev Ding Buffer*⟩≡
```
Uint8 dingBuf[ FREQ ];
unsigned int dingLen;
```

4.1 The Constructor and Destructor

The default constructor is the only one available for the sound device. It, of course, takes no arguments.

⟨*SoundDev Constructor Declaration*⟩≡
```
SoundDev( void );
```

The constructor for the sound device attempts to open the sound device for a single channel of the desired frequency. It first has to prepare the desired audio specification. Then, it has to attempt to open the device itself. If the device was opened, then we'll prepare a buffer with a ding noise in it.

⟨*SoundDev Constructor Implementation*⟩≡

```
⟨SoundDevNameSpace⟩::SoundDev( void )
        : currentBuf( 0 ), currentPtr( 0 ), currentLen( 0 )
{
    ⟨SoundDev Prepare Desired Spec⟩
    ⟨SoundDev Open Audio⟩

    if ( this->opened ) {
        ⟨SoundDev fill ding buffer⟩
    }
}
```

The desired sound format is 8-bit signed data at 8KHz with a sixty-fourth of a second buffer.

⟨*SoundDev Prepare Desired Spec*⟩≡

```
    SDL_AudioSpec desired;

    desired.freq = FREQ;
    desired.format = AUDIO_S16SYS;
    desired.channels = 1;
    desired.samples = ( FREQ >> 6 );
    desired.callback = SoundDev::callbackTrampoline;
    desired.userdata = this;
```

To open the audio device, we simply call the appropriate method from the SDL library. If we didn't get the frequency that we wanted, then we will just close up and forget about audio.

⟨*SoundDev Open Audio*⟩≡

```
    if ( ::SDL_OpenAudio( &desired, &this->spec ) < 0 ) {
        this->opened = false;
    } else {
        if ( this->spec.freq != FREQ
        || this->spec.format != AUDIO_S16SYS ) {
            ::SDL_CloseAudio();
            this->opened = false;
        } else {
            this->opened = true;
        }
    }
```

First, we reset the length of the ding buffer to zero. Then, we fill in the first part of the buffer with our tone getting louder and louder from zero up to its full volume of 8192 out of 32767 in the first 64-th of a second. Then, we let the sound decay from that maximum volume. We let it decay really quickly to give the ding a little bit of pop.

⟨*SoundDev fill ding buffer*⟩≡
```
this->dingLen = 0;
double volume = 0.0;
Uint16* ptr = (Uint16*)&this->dingBuf[ 0 ];

while ( this->dingLen < FREQ && volume < 8192.0 ) {
    ⟨SoundDev add to ding⟩
    volume += 8192.0 * 64.0 / (double)FREQ;
}

while ( this->dingLen < FREQ && volume > 1.0 ) {
    ⟨SoundDev add to ding⟩
    volume *= 0.90;
}
```

The sample at any given point is the based upon the frequency and the volume. We're constructing a 256Hz tone. Then, we clip that tone into the range of a valid 16-bit number. Then, we add the sample into the buffer.

⟨*SoundDev add to ding*⟩≡
```
double angle = ( (double)this->dingLen / (double)FREQ * M_PI * 512.0 );
int ival = (int)( ::sin( angle ) * volume );

⟨SoundDev clip value⟩

unsigned short val = ( ival & 0x00FFFF );
*ptr++ = val;
this->dingLen += sizeof( Uint16 );
```

The valid range for a 16-bit number is from negative 32768 to negative 32767.

⟨*SoundDev clip value*⟩≡
```
if ( ival >= 32767 ) {
    ival = 32767;
} else if ( ival < -32768 ) {
    ival = -32768;
}
```

The destructor for the sound device has to release the audio device with SDL.

⟨*SoundDev Destructor Declaration*⟩≡
```
~SoundDev( void );
```

The destructor for the sound device has to release the audio device with SDL. But, it only has to do this if the device is still open. It also has to release any buffer that is currently around.

⟨*SoundDev Destructor Implementation*⟩≡

```
⟨SoundDevNameSpace⟩::~SoundDev( void )
{
    if ( this->opened ) {
        ::SDL_PauseAudio( 1 );
        ::SDL_CloseAudio();
    }

    delete[] this->currentBuf;
}
```

4.2 Checking The Sound Device

Currently, the only check one can do on the sound device is see if it is opened. This is a simple inline function because it requires almost nothing for code.

⟨*SoundDev Is Opened Inline*⟩≡

```
inline bool isOpened( void ) const {
    return this->opened;
};
```

4.3 Playing A Buffer

This method takes a buffer that already contains a sound and prepares it to be played on the audio device.

⟨*SoundDev Play Declaration*⟩≡

```
void play( Uint8* buffer, unsigned int len );
```

This method pauses the audio that is currently playing. Then, it deletes the current buffer. Then, it allocates a new buffer and copies the parameters into the sound device. Then, it restarts the audio.

⟨*SoundDev Play Implementation*⟩≡

```
void
⟨SoundDevNameSpace⟩::play( Uint8* buffer, unsigned int len )
{
    ::SDL_PauseAudio( 1 );

    delete[] this->currentBuf;

    this->currentBuf = new Uint8[ len ];
    ::memcpy( this->currentBuf, buffer, len );

    this->currentPtr = this->currentBuf;
    this->currentLen = len;

    ::SDL_PauseAudio( 0 );
}
```

4.4 Making A Ding

This method generates a ding noise and plays it.

⟨*SoundDev Ding Declaration*⟩≡

```
void ding( void );
```

This method sets up a buffer big enough for the ding noise. Then, it fills in the buffer with data for the noise. The "ding" is a quickly rising pulse of a 256Hz tone that falls off in volume exponentially. Then, it calls the `play()` method defined above to play the noise.

⟨*SoundDev Ding Implementation*⟩≡

```
void
⟨SoundDevNameSpace⟩::ding( void )
{
    this->play( this->dingBuf, this->dingLen );
}
```

4.5 The Callback Functions

There are two callback functions defined in the sound device class. The one that does all of the work for the class is an instance method.

⟨*SoundDev Callback Declarations*⟩≡

```
void callback( Uint8* stream, int len );
```

The other method defined simply fields the callback from the SDL library and converts it into a method call on the instance.

⟨*SoundDev Callback Declarations*⟩+≡

```
static void callbackTrampoline(
        void* userData, Uint8* stream, int len
    );
```

Currently, the callback function tries to copy data out of the current buffer if there is some. If there is not any data left in the current buffer, then it simply pauses the audio.

⟨*SoundDev Callback Implementations*⟩≡

```
void
⟨SoundDevNameSpace⟩::callback( Uint8* stream, int len )
{
    if ( this->currentLen > 0 ) {
        unsigned int copyLen = this->currentLen;

        if ( copyLen > len ) {
            copyLen = len;
        }

        ::memcpy( stream, this->currentPtr, copyLen );
        this->currentPtr += copyLen;
        this->currentLen -= copyLen;

        stream += copyLen;
        len -= copyLen;
    } else {
        ::SDL_PauseAudio( 1 );
    }

    if ( len > 0 ) {
        ::memset( stream, 0, len );
    }
}
```

As mentioned above, the callback trampoline class-method simply springs from the SDL callback into a method call on the instance given as user data.

⟨*SoundDev Callback Implementations*⟩+≡

```
void
⟨SoundDevNameSpace⟩::callbackTrampoline(
        void* userData, Uint8* stream, int len
    )
{
    assert( userData != 0 );
    SoundDev* dev = (SoundDev*)userData;
    dev->callback( stream, len );
}
```

4.6 The `SoundDev` class

In this section, we assemble the `SoundDev` class from the pieces in the sections above.

First, we include constant declarations so that they will be readily available for use in other declarations.

⟨*SoundDev Class Definition*⟩≡

```
public:
    ⟨SoundDev Frequency⟩
```

Then, we declare the constructor and destructor for the sound device class.

⟨*SoundDev Class Definition*⟩+≡

```
public:
    ⟨SoundDev Constructor Declaration⟩
    ⟨SoundDev Destructor Declaration⟩
```

After the constructor, we declare the method used to check if the device is opened.

⟨*SoundDev Class Definition*⟩+≡

```
public:
    ⟨SoundDev Is Opened Inline⟩
```

Then, we declare the methods used to actually play sounds.

⟨*SoundDev Class Definition*⟩+≡

```
protected:
    ⟨SoundDev Play Declaration⟩
public:
    ⟨SoundDev Ding Declaration⟩
```

After that, we declare the callback-related functions.

⟨*SoundDev Class Definition*⟩+≡

```
private:
    ⟨SoundDev Callback Declarations⟩
```

Next, we declare the member variables used to hold the state of the device that was opened and any buffer currently being played.

⟨*SoundDev Class Definition*⟩+≡
```
private:
    ⟨SoundDev Audio Spec⟩
    ⟨SoundDev Current Buffer⟩
```

Next, we declare the member variables which contain particular sounds.

⟨*SoundDev Class Definition*⟩+≡
```
    ⟨SoundDev Ding Buffer⟩
```

Once these declarations are all done, we throw all of these together into the class declaration itself.

⟨*SoundDev Class Declaration*⟩≡
```
class SoundDev {
    ⟨SoundDev Class Definition⟩
};
```

4.7 The `soundDev.h` file

In this section, we assemble the header file for the `SoundDev` class. It is really straightforward since we assembled the class declaration in the previous section. The only thing that we add to the class declaration is that we tuck it into our own name space so that we can keep the global namespace squeaky clean.

⟨*soundDev.h*⟩≡
```
namespace ⟨NameSpace⟩ {
    ⟨SoundDev Class Declaration⟩
};
```

4.8 The `soundDev.cpp` file

For the actual C++ source code, we include the header file that defines `assert()`, the one that defines `memset()`, the SDL headers needed, and then include the header file generated in the previous section.

⟨*soundDev.cpp*⟩≡
```
#include <assert.h>
#include <string.h>
#include <math.h>
#include <SDL.h>
#include "soundDev.h"

#ifndef M_PI
 #define M_PI 3.14159
#endif
```

After that, we incorporate the implementations of the constructor and destructor.

⟨*soundDev.cpp*⟩+≡
⟨*SoundDev Constructor Implementation*⟩
⟨*SoundDev Destructor Implementation*⟩

Then, we incorporate the implementations of the methods used to play sounds.

⟨*soundDev.cpp*⟩+≡
⟨*SoundDev Play Implementation*⟩
⟨*SoundDev Ding Implementation*⟩

The source file also includes the code for the callback routines.

⟨*soundDev.cpp*⟩+≡
⟨*SoundDev Callback Implementations*⟩

5 The Generic Game Controller

So that the main loop doesn't have to deal with five different game controllers, each of the game controllers inherits from this base class. This class gives the main loop a method to call with mouse events. Also, it gives the `View` class a consistent interface to call when buttons on the interface are pressed. And, it includes variables which will be common to all of the game controllers.

The namespace inside the controller class is a concatenation of the general namespace and the name of the controller class.

⟨*ControllerNameSpace*⟩≡
⟨*NameSpace*⟩`::Controller`

The `Controller` class keeps a pointer to the cube used for the game.

⟨*Controller Cube*⟩≡
```
Cube* cube;
```

The `Controller` class also tracks the number of dimensions that are being used. It needs this information so that it can properly determine the screen coordinate to cell coordinate transformations.

⟨*Controller Dimensions*⟩≡
```
unsigned int dims;
```

And, the `Controller` class tracks the current skill level so that it knows which buttons should be displayed "pressed".

⟨*Controller Skill Level*⟩≡
```
unsigned int skillLevel;
```

The `Controller` class also keeps track of whether or not it is wrapping around so that it can properly render that button, too.

⟨*Controller Wrap*⟩≡
```
bool wrap;
```

5.1 The Constructor and Destructor

The constructor for the `Controller` class takes four arguments. The first is a pointer to the game cube, the second specifies the number of dimensions to employ, the third specifies the skill level to use, and the last specifies whether the edges wrap around.

⟨*Controller Constructor Declaration*⟩≡
```
Controller(
        Cube* _cube,
        unsigned int _dims = 2,
        unsigned int _skillLevel = 0,
        bool _wrap = true
    );
```

The constructor for the `Controller` class simply copies the arguments into its local variables. Then, it does some simple checks on them to ensure that they meet expectations.

⟨*Controller Constructor Implementation*⟩≡

```
⟨ControllerNameSpace⟩::Controller(
        Cube* _cube,
        unsigned int _dims,
        unsigned int _skillLevel,
        bool _wrap
    ) : cube( _cube ),
        dims( _dims ),
        skillLevel( _skillLevel ),
        wrap( _wrap )
{
    assert( cube != 0 );
    assert( dims > 1 );
    assert( dims <= ⟨CubeNameSpace⟩::DIMENSIONS );
    assert( skillLevel < 3 );
}
```

The destructor for the `Controller` class doesn't have to do anything. It is merely a place-holder to ensure that instances of subclasses get destructed properly.

⟨*Controller Destructor Declaration*⟩≡

```
virtual ~Controller( void );
```

⟨*Controller Destructor Implementation*⟩≡

```
⟨ControllerNameSpace⟩::~Controller( void )
{
}
```

5.2 The Mouse Event Interface

The routine which handles mouse events needs to know whether the event is a mouse press or mouse release. It also needs to know where the event happened. And, it needs to know which mouse button was pressed.

⟨*Controller Mouse Click Interface*⟩≡

```
virtual void handleMouseClick(
        bool isMouseUp,
        unsigned int xx,
        unsigned int yy,
        unsigned int buttonNumber
    ) = 0;
```

5.3 The Game Setting Interface

So that the view buttons can easily affect the controller state, the controller contains methods which allow the view class to tweak the game parameters. These are pure virtual methods here but will be overridden in the derived classes to make the appropriate changes to the actual game-model class.

⟨*Controller Game Setting Interface*⟩≡
```
virtual void setDimension( unsigned int _dims ) = 0;
virtual void setSkillLevel( unsigned int _skillLevel ) = 0;
virtual void setWrap( bool _wrap ) = 0;
virtual void newGame( void ) = 0;
```

5.4 The `Controller` class

In this section, we assemble the `Controller` class from the pieces in the sections above.

The controller class needs its constructor and destructor.

⟨*Controller Class Definition*⟩≡
```
protected:
    ⟨Controller Constructor Declaration⟩
public:
    ⟨Controller Destructor Declaration⟩
```

We include, in the `Controller` class, the interface used for mouse clicks.

⟨*Controller Class Definition*⟩+≡
```
public:
    ⟨Controller Mouse Click Interface⟩
```

The `Controller` class also defines the methods used by the `View` class to update the settings for the game.

⟨*Controller Class Definition*⟩+≡
```
public:
    ⟨Controller Game Setting Interface⟩
```

The `Controller` class also includes its member variables

⟨*Controller Class Definition*⟩+≡
```
protected:
    ⟨Controller Cube⟩
    ⟨Controller Dimensions⟩
    ⟨Controller Skill Level⟩
    ⟨Controller Wrap⟩
```

Once these declarations are all done, we throw all of these together into the class declaration itself.

⟨*Controller Class Declaration*⟩≡
```
class Controller {
    ⟨Controller Class Definition⟩
};
```

5.5 The `controller.h` file

In this section, we assemble the header file for the `Controller` class. It is really straightforward since we assembled the class declaration in the previous section. The only thing that we add to the class declaration is that we tuck it into our own name space so that we can keep the global namespace squeaky clean.

⟨*controller.h*⟩≡

```
namespace ⟨NameSpace⟩ {
    ⟨Controller Class Declaration⟩
};
```

5.6 The `controller.cpp` file

There is not much to the `controller.cpp` source file at the moment. It only contains the source code for the constructor and the destructor.

⟨*controller.cpp*⟩≡

```
#include <assert.h>
#include "cube.h"
#include "controller.h"

⟨Controller Constructor Implementation⟩
⟨Controller Destructor Implementation⟩
```

6 The Generic Game View

So that the controllers don't have to each deal with different game views, each of the game views inherits from this base class. This class gives the controllers a method to draw the screen. And, it gives the controllers a method to use to convert screen coordinates to cell coordinates.

The namespace inside the view class is a concatenation of the general namespace and the name of the view class.

⟨*ViewNameSpace*⟩≡
⟨*NameSpace*⟩`::View`

The `View` class contains a pointer to the screen for output

⟨*View Screen*⟩≡
```
SDL_Surface* screen;
```

The `View` class contains a pointer to the sound device for audio output.

⟨*View Sound*⟩≡
```
SoundDev* sound;
```

The `View` class contains a pointer to the current game cube.

⟨*View Cube*⟩≡
```
Cube* cube;
```

The `View` class tracks the number of dimensions so that this information is available for the screen coordinate to cell index transformations. And, it keeps track of the skill level and wrapping mode so that it can update the buttons properly.

⟨*View Game State*⟩≡
```
unsigned int dims;
unsigned int skillLevel;
bool wrap;
```

The `View` class contains a pointer to the image for the backdrop of the sidebar. It also contains a pointer to the image for the text-overlay that is painted over the buttons. It contains a pointer to the images for up and down dimensions buttons. It contains a pointer to the images for up and down settings buttons. It contains a pointer to the images for up and down game-action buttons. And, It contains a pointer to the image for up and down help buttons.

⟨*View Images*⟩≡
```
SDL_Surface* sidebar;
SDL_Surface* overlay;
SDL_Surface* dimButton[2];
SDL_Surface* setButton[2];
SDL_Surface* actButton[2];
SDL_Surface* helpButton[2];
```

In addition to these images for the sidebar, the `View` class also holds a pointer to the image used to congratulate the winner and a pointer to the image used to harrass the loser when a game has been completed.

⟨ *View Images* ⟩+≡
```
SDL_Surface* victory;
SDL_Surface* losing;
```

The `View` class also keeps track of which buttons should be pressed and which should be unpressed. But, first, it must define which buttons there are.

⟨ *View Button Names* ⟩≡
```
enum {
    DIM_2 = 0,
    DIM_3,
    DIM_4,
    EASY,
    MEDIUM,
    HARD,
    WRAP,
    NEW,
    BACK,
    HELP,
    MAX_BUTTON
};
```

After that, it can use that information to just keep an array of booleans for whether the button is pressed or not.

⟨ *View Button States* ⟩≡
```
bool bPressed[ MAX_BUTTON ];
```

The `View` also keeps track of when an item has been clicked down but the mouse has not yet been released. This is used for visual feedback to the user. The `View` also tracks which state the button was in before the person pressed it.

⟨ *View Button States* ⟩+≡
```
unsigned int clickedButton;
bool originalState;
```

It also keeps a static array of where the buttons are located.

⟨ *View Sidebar Location* ⟩≡
```
enum { SIDEBAR_X = 600, SIDEBAR_Y = 0 };
```

⟨ *View Button Location* ⟩≡
```
static SDL_Rect bLocation[ MAX_BUTTON ];
```

⟨*View Button Location Declaration*⟩≡
```
SDL_Rect ⟨ViewNameSpace⟩::bLocation[
        ⟨ViewNameSpace⟩::MAX_BUTTON
    ] =
{
    { SIDEBAR_X + 2, SIDEBAR_Y + 2, 64, 64 },
    { SIDEBAR_X + 68, SIDEBAR_Y + 2, 64, 64 },
    { SIDEBAR_X + 134, SIDEBAR_Y + 2, 64, 64 },
    { SIDEBAR_X + 2, SIDEBAR_Y + 68, 64, 32 },
    { SIDEBAR_X + 68, SIDEBAR_Y + 68, 64, 32 },
    { SIDEBAR_X + 134, SIDEBAR_Y + 68, 64, 32 },
    { SIDEBAR_X + 68, SIDEBAR_Y + 102, 64, 32 },
    { SIDEBAR_X + 36, SIDEBAR_Y + 180, 128, 64 },
    { SIDEBAR_X + 36, SIDEBAR_Y + 250, 128, 64 },
    { SIDEBAR_X + 2, SIDEBAR_Y + 534, 196, 64 }
};
```

The `View` class keeps track of the color that it will paint the background of the cube area.

⟨*View Colors*⟩≡
```
unsigned int bgColor;
```

The `View` class defines several constants for use in its internal calculations. It defines `SQUARE` to be the size of each displayed cell of the cube.

⟨*View Constant Definitions*⟩≡
```
enum { SQUARE = 36 };
```

It defines `GAP` to be the size of the gap between each two-dimensional array of cells.

⟨*View Constant Definitions*⟩+≡
```
enum { GAP = 4 };
```

It defines `BLOCK` to be the spacing between adjacent two-dimensional arrays of cells.

⟨*View Constant Definitions*⟩+≡
```
enum { BLOCK = ( ⟨CubeNameSpace⟩::SIDE_LENGTH * SQUARE ) + GAP };
```

The `View` also defines an array that tracks the starting screen coordinates of cubes of different dimensions.

⟨*View Cube Start Coordinates*⟩≡
```
static unsigned int startCoords[
        ⟨CubeNameSpace⟩::DIMENSIONS + 1
    ][ 2 ];
```

⟨*View Cube Start Coordinates Definition*⟩≡
```
unsigned int ⟨ViewNameSpace⟩::startCoords[
        ⟨CubeNameSpace⟩::DIMENSIONS + 1
    ][ 2 ] = {
        { 0, 0 },
        {
            ( 600 - ( BLOCK - GAP ) ) / 2,
            ( 600 - ( SQUARE ) ) / 2
        },
        {
            ( 600 - ( BLOCK - GAP ) ) / 2,
            ( 600 - ( BLOCK - GAP ) ) / 2
        },
        {
            ( 600 - ( ⟨CubeNameSpace⟩::SIDE_LENGTH * BLOCK - GAP ) ) / 2,
            ( 600 - ( BLOCK - GAP ) ) / 2
        },
        {
            ( 600 - ( ⟨CubeNameSpace⟩::SIDE_LENGTH * BLOCK - GAP ) ) / 2,
            ( 600 - ( ⟨CubeNameSpace⟩::SIDE_LENGTH * BLOCK - GAP ) ) / 2
        }
    };
```

The view also keeps a pointer to the current `Help` context if there is one.

⟨*View Help*⟩≡
```
    Help* help;
```

6.1 The Constructor

The constructor for the generic view class takes six arguments. The first is a pointer to the screen, the second is a pointer to the sound device, the third is a pointer to the game cube, the fourth specifies the number of dimensions to employ, the fifth specifies the skill level, and the sixth specifies the wrapping mode.

⟨*View Constructor Declaration*⟩≡
```
    View(
            SDL_Surface* _screen,
            SoundDev* _sound,
            Cube* _cube,
            unsigned int _dims = 2,
            unsigned int _skillLevel = 0,
            bool _wrap = true
        );
```

The constructor copies the arguments into local variables. Then, it attempts to prepare the background color for the cube and to load the images for the sidebar and buttons. Then, it calls its own `reset()` method to update the button pressed states and draw the screen.

⟨*View Constructor Implementation*⟩≡

```
⟨ViewNameSpace⟩::View(
        SDL_Surface* _screen,
        SoundDev* _sound,
        Cube* _cube,
        unsigned int _dims,
        unsigned int _skillLevel,
        bool _wrap
    ) : screen( _screen ), sound( _sound ), cube( _cube ),
        dims( _dims ), skillLevel( _skillLevel ), wrap( _wrap ),
        help( 0 )
{
    if ( this->screen != 0 ) {
        SDL_PixelFormat* fmt = this->screen->format;
        this->bgColor = SDL_MapRGB( fmt, 0, 0, 0 );
    }

    ⟨View Load Images⟩

    this->clickedButton = MAX_BUTTON;
    this->reset();
}
```

The view class loads the image for the backdrop of the sidebar and the text overlay that is displayed on top of the buttons.

⟨*View Load Images*⟩≡

```
this->sidebar = ::IMG_Load( "../../data/panel.png" );
this->overlay = ::IMG_Load( "../../data/overlay.png" );
```

Then, it loads the two states for the dimension buttons.

⟨*View Load Images*⟩+≡

```
this->dimButton[ 0 ] = ::IMG_Load( "../../data/dimOff.png" );
this->dimButton[ 1 ] = ::IMG_Load( "../../data/dimOn.png" );
```

Then, it loads the two states for the settings buttons.

⟨*View Load Images*⟩+≡

```
this->setButton[ 0 ] = ::IMG_Load( "../../data/setOff.png" );
this->setButton[ 1 ] = ::IMG_Load( "../../data/setOn.png" );
```

Then, it loads the two states for the game-action buttons.

⟨*View Load Images*⟩+≡

```
this->actButton[ 0 ] = ::IMG_Load( "../../data/actOff.png" );
this->actButton[ 1 ] = ::IMG_Load( "../../data/actOn.png" );
```

Then, it loads the two states for the help button.

⟨ *View Load Images* ⟩+≡
```
this->helpButton[ 0 ] = ::IMG_Load( "../../data/helpOff.png" );
this->helpButton[ 1 ] = ::IMG_Load( "../../data/helpOn.png" );
```

Finally, it loads the images to display when the game has been completed.

⟨ *View Load Images* ⟩+≡
```
this->victory = ::IMG_Load( "../../data/victory.png" );
this->losing = ::IMG_Load( "../../data/defeat.png" );
```

The destructor for the view must release the images loaded above.

⟨ *View Destructor Declaration* ⟩≡
```
virtual ~View( void );
```

⟨ *View Destructor Implementation* ⟩≡
```
⟨ ViewNameSpace ⟩::~View( void )
{
    ⟨ View Release Images ⟩
}
```

First, the destructor releases the memory for the messages that are displayed at the end of a game.

⟨ *View Release Images* ⟩≡
```
::SDL_FreeSurface( this->losing );
::SDL_FreeSurface( this->victory );
```

Next, the destructor releases the two states for the help button.

⟨ *View Release Images* ⟩+≡
```
::SDL_FreeSurface( this->helpButton[ 1 ] );
::SDL_FreeSurface( this->helpButton[ 0 ] );
```

Then, it release the two states for the game-action buttons.

⟨ *View Release Images* ⟩+≡
```
::SDL_FreeSurface( this->actButton[ 1 ] );
::SDL_FreeSurface( this->actButton[ 0 ] );
```

Then, it release the two states for the settings buttons.

⟨ *View Release Images* ⟩+≡
```
::SDL_FreeSurface( this->setButton[ 1 ] );
::SDL_FreeSurface( this->setButton[ 0 ] );
```

Then, it release the two states for the dimension buttons.

⟨ *View Release Images* ⟩+≡
```
::SDL_FreeSurface( this->dimButton[ 1 ] );
::SDL_FreeSurface( this->dimButton[ 0 ] );
```

Finally, the view class releases the sidebar image and the text overlay.

⟨ *View Release Images* ⟩+≡
```
::SDL_FreeSurface( this->overlay );
::SDL_FreeSurface( this->sidebar );
```

6.2 Resetting the Button States

This method is used to reset the button states of all of the buttons.

⟨*View Reset Declaration*⟩≡
```
void reset( void );
```

This method turns off all of the buttons and then turns on the appropriate dimension and settings buttons.

⟨*View Reset Implementation*⟩≡
```
void
⟨ViewNameSpace⟩::reset( void )
{
    for ( unsigned int ii=0; ii < MAX_BUTTON; ++ii ) {
        this->bPressed[ ii ] = false;
    }

    this->bPressed[ DIM_2 + dims - 2 ] = true;
    this->bPressed[ EASY + this->skillLevel ] = true;
    this->bPressed[ WRAP ] = this->wrap;
}
```

6.3 Handling Mouse Clicks

When the controller receives a mouse click, it passes it on to the `View` class. The view class is responsible for determining if any of the buttons in the sidebar were clicked. If they were, then the appropriate update method is invoked on the controller.

⟨*View Mouse Click Interface*⟩≡
```
virtual bool handleMouseClick(
        Controller* controller,
        bool isMouseUp,
        unsigned int xx,
        unsigned int yy,
        unsigned int buttonNumber
    );
```

If the event is a "MouseDown" event, this method loops through each button to see which button (if any) was clicked. If a button was clicked, this is stored in the `clickedButton` member. On a mouse up event where the `clickedButton` member had been set, the button is set back to its original state. Then, if the release was still inside the button, the appropriate action for that button takes place.

⟨*View Mouse Click Implementation*⟩≡

```
bool
⟨ViewNameSpace⟩::handleMouseClick(
        Controller* control,
        bool isMouseUp,
        unsigned int xx,
        unsigned int yy,
        unsigned int buttonNumber
    )
{
    ⟨View Mouse Click check for help-mode click⟩

    if ( ! isMouseUp ) {
        for ( unsigned int ii=0; ii < MAX_BUTTON; ++ii ) {
            if ( this->checkButton( xx, yy, ii ) ) {
                this->clickedButton = ii;
                ⟨View Mouse press button⟩
                return true;
            }
        }
    } else if ( isMouseUp && this->clickedButton < MAX_BUTTON ) {
        unsigned int ii = this->clickedButton;
        bool inSide = this->checkButton( xx, yy, ii );

        ⟨View Mouse unpress button⟩

        if ( inSide ) {
            ⟨View Mouse Click do button thing⟩
        } else {
            this->drawButton( ii );
        }

        return true;
    }

    return false;
}
```

If we're in help mode, then we have to careful to let the help mode handle any clicks before we do.

⟨*View Mouse Click check for help-mode click*⟩≡

```
if ( this->help != 0 ) {
    bool hit = this->help->handleMouseClick(
            isMouseUp, xx, yy, buttonNumber
        );
    if ( hit ) {
        return true;
    }
}
```

When pressing the button, we save the original state so that it can be restored when we're finished. Then, we redraw the button in the opposite state.

⟨*View Mouse press button*⟩≡

```
this->originalState = this->bPressed[ this->clickedButton ];
if ( ! this->originalState ) {
    this->bPressed[ this->clickedButton ] = true;
    this->drawButton( this->clickedButton );
}
```

When we go to unpress the button, we set it back to its original state. Then, we clear the fact that this button has been clicked.

⟨*View Mouse unpress button*⟩≡

```
this->bPressed[ this->clickedButton ] = this->originalState;
this->clickedButton = MAX_BUTTON;
```

The proper thing to do when the mouse is released in the button that it was clicked in, depends upon which button that is. First off, any click on a button will bump us out of help mode. For the dimension buttons, we reset the dimensions. For the skill level buttons, we reset the skill. For the wrap button, we toggle it. For the new game button, we restart the current game. For the back button, we kick off an event to load the main menu. For any other button, we just redraw the button.

⟨*View Mouse Click do button thing*⟩≡

```
if ( this->help != 0 ) {
    this->setHelp( 0 );
    this->drawButton( ii );
    this->drawButton( HELP );
    this->redraw();
} else if ( ii >= DIM_2 && ii <= DIM_4 ) {
    this->dims = ii + 2 - DIM_2;
    control->setDimension( this->dims );
} else if ( ii >= EASY && ii <= HARD ) {
    this->skillLevel = ii - EASY;
    control->setSkillLevel( this->skillLevel );
} else if ( ii == WRAP ) {
    this->wrap = ! this->wrap;
    control->setWrap( this->wrap );
} else if ( ii == NEW ) {
    control->newGame();
} else if ( ii == BACK ) {
    SDL_Event change;
    change.type = SDL_USEREVENT;
    change.user.code = -1;
    ::SDL_PushEvent( &change );
} else if ( ii == HELP ) {
    this->setHelp( new Help( this, this->screen ) );
} else {
    this->drawButton( ii );
}
```

The mouse click function often needs to check whether a point is inside the bounding box for a particular button. This method does that so that the same code doesn't have to be compiled twice.

⟨*View Mouse Check Button Declaration*⟩≡

```
bool checkButton(
        unsigned int xx, unsigned int yy,
        unsigned int ii
    );
```

The implementation is straightforward. It simply checks to make sure that both the x- and y-coordinates are within the box.

⟨*View Mouse Check Button Implementation*⟩≡

```
bool
⟨ViewNameSpace⟩::checkButton(
        unsigned int xx, unsigned int yy,
        unsigned int ii
    )
{
    return ( xx >= this->bLocation[ ii ].x
        && xx < this->bLocation[ ii ].x + this->bLocation[ ii ].w
        && yy >= this->bLocation[ ii ].y
        && yy < this->bLocation[ ii ].y + this->bLocation[ ii ].h
    );
}
```

6.4 Converting Between Screen and Cell Coordinates

Most controllers will need to go from mouse coordinates to cube coordinates and back again. These two methods facilitate that.

The first method converts from screen coordinates `xx` and `yy` into a cell index `index`. It returns false if the click missed the cube.

⟨*View Screen-Cell Declarations*⟩≡

```
static bool screenToCell(
        unsigned int xx, unsigned int yy,
        unsigned int dims,
        unsigned int* index
    );
```

This method first ensures that the `index` pointer is valid and that the number of dimensions can be dealt with by this routine. Then, it resets the coordinates based upon where the top corner of the cube is to be displayed. Then, it determines the cell coordinates for that point. Then, it makes sure that the cell coordinates are valid. Then, it gets the index from the `Cube` class method.

⟨*View Screen-Cell Implementations*⟩≡

```
bool
⟨ViewNameSpace⟩::screenToCell(
        unsigned int xx, unsigned int yy,
        unsigned int dims,
        unsigned int* index
    )
{
    assert( index != 0 );
    assert( dims <= 4 && dims > 0 );

    ⟨View Screen-Cell Reset Origin⟩
    ⟨View Screen-Cell Prepare Cell Coords⟩
    ⟨View Screen-Cell Check Cell Coords⟩

    ⟨CubeNameSpace⟩::vectorToIndex( coords, index );
    return true;
}
```

To reset the origin, we subtract out the starting coordinates for the current number of dimensions. If this number would be negative, then we bail out.

⟨*View Screen-Cell Reset Origin*⟩≡

```
unsigned int sx = startCoords[ dims ][ 0 ];
unsigned int sy = startCoords[ dims ][ 1 ];

if ( xx < sx || sy < sy ) {
    return false;
}

xx -= sx;
yy -= sy;
```

To get the cell coordinates, we have to take into account the fact that there may be multiple boards in a row. I believe the following code is the most straightforward. The position within each tier is the screen-position modulo the block size divided by the size of each cell. Which tier is simply the position divided by the block size.

⟨*View Screen-Cell Prepare Cell Coords (clear version)*⟩≡
```
unsigned int coords[ ⟨CubeNameSpace⟩::DIMENSIONS ];

coords[ 0 ] = ( xx % BLOCK ) / SQUARE;
coords[ 1 ] = ( yy % BLOCK ) / SQUARE;
coords[ 2 ] = xx / BLOCK;
coords[ 3 ] = yy / BLOCK;
```

However, that method requires six integer divisions. Of course, if the compiler were really snazzy, it could determine `xx % BLOCK` and `xx / BLOCK` simultaneously on many architectures. However, integer division takes far longer on most processors than integer multiplication. For example, on the Pentium, division of two unsigned 32-bit integers takes 41 clock cycles. On the same processor, with the same size operands, multiplication takes 10 clock cycles and subtraction takes 3 clock cycles. Thus, I'm actually going to use the following bit of code, which should do the same thing with only four integer divisions.

⟨*View Screen-Cell Prepare Cell Coords*⟩≡
```
unsigned int coords[ ⟨CubeNameSpace⟩::DIMENSIONS ];

coords[ 2 ] = xx / BLOCK;
coords[ 3 ] = yy / BLOCK;

coords[ 0 ] = ( xx - coords[ 2 ] * BLOCK ) / SQUARE;
coords[ 1 ] = ( yy - coords[ 3 ] * BLOCK ) / SQUARE;
```

Then, we have to check to make sure that none of the coordinates in the dimensions of interest exceeded the length of a side of the cube.

⟨*View Screen-Cell Check Cell Coords*⟩≡
```
for ( unsigned int ii=0; ii < dims; ++ii ) {
    if ( coords[ ii ] >= ⟨CubeNameSpace⟩::SIDE_LENGTH ) {
        return false;
    }
}
```

And, we have to check that all of the coordinates for higher dimensions than those of interest came out to be zero.

⟨*View Screen-Cell Check Cell Coords*⟩+≡

```
for ( unsigned int ii=dims; ii < ⟨CubeNameSpace⟩::DIMENSIONS; ++ii ) {
    if ( coords[ ii ] > 0 ) {
        return false;
    }
}
```

The second method gets the screen coordinates `xx` and `yy` of the upper-left corner of the cell with a given index `index`.

⟨*View Screen-Cell Declarations*⟩+≡

```
static void cellToScreen(
        unsigned int index,
        unsigned int dims,
        unsigned int* xx, unsigned int* yy
    );
```

This method is significantly easier than the previous method. It simply needs to get the coordinates for the index from the cube and then use those to determine the screen coordinates.

⟨*View Screen-Cell Implementations*⟩+≡

```
void
⟨ViewNameSpace⟩::cellToScreen(
        unsigned int index,
        unsigned int dims,
        unsigned int* xx, unsigned int* yy
    )
{
    assert( xx != 0 );
    assert( yy != 0 );

    unsigned int coords[ ⟨CubeNameSpace⟩::DIMENSIONS ];
    ⟨CubeNameSpace⟩::indexToVector( index, coords );

    *xx = startCoords[ dims ][ 0 ]
        + coords[ 0 ] * SQUARE
        + coords[ 2 ] * BLOCK
        ;

    *yy = startCoords[ dims ][ 1 ]
        + coords[ 1 ] * SQUARE
        + coords[ 3 ] * BLOCK
        ;
}
```

6.5 The Redraw Methods

The view class has a method which allows one to update the entire display area for the game.

⟨*View Redraw Declarations*⟩≡
```
virtual void redraw( void );
```

The redraw function for the base view simply blanks out the backdrop of the cube area and draws the sidebar with its buttons and its text overlay.

⟨*View Redraw Implementation*⟩≡
```
void
⟨ViewNameSpace⟩::redraw( void )
{
    SDL_Rect rect;

    ⟨View Blank Cube Area⟩
    ⟨View Draw Sidebar⟩
    ⟨View Draw Buttons⟩
    ⟨View Draw Overlay⟩
}
```

To blank the cube area, this method simply creates a rectangle the size of the whole cube area and fills it with the background color.

⟨*View Blank Cube Area*⟩≡
```
rect.x = 0;
rect.y = 0;
rect.w = 600;
rect.h = 600;
::SDL_FillRect( this->screen, 0, this->bgColor );
```

To draw the sidebar, this method simply copies the sidebar to the right-hand portion of the screen.

⟨*View Draw Sidebar*⟩≡
```
if ( this->sidebar != 0 ) {
    rect.x = SIDEBAR_X;
    rect.y = SIDEBAR_Y;
    rect.w = 200;
    rect.h = 600;
    ::SDL_BlitSurface( this->sidebar, 0, this->screen, &rect );
}
```

The `View` class runs through each of its buttons. It calls its own `drawButton` method for each one. It passes in the index for the button. And, it sets the update flag to `false` here because the buttons will all be refreshed at once when the panel has been completely drawn.

⟨*View Draw Buttons*⟩≡
```
for ( unsigned int ii=0; ii < MAX_BUTTON; ++ii ) {
    this->drawButton( ii, false );
}
```

To draw the overlay, this method simply copies the overlay to the right-hand portion of the screen.

⟨*View Draw Overlay*⟩≡
```
if ( this->overlay != 0 ) {
    rect.x = SIDEBAR_X;
    rect.y = SIDEBAR_Y;
    rect.w = 200;
    rect.h = 600;
    ::SDL_BlitSurface( this->overlay, 0, this->screen, &rect );
}
```

The view class has a method which allows one to update a single cell of the cube by index. The base class does not implement this method itself because each game will have cells which look different depending on the state of the game cube.

⟨*View Redraw Declarations*⟩+≡
```
virtual void redraw( unsigned int index ) = 0;
```

The view class has a method that redraws a single button in the side-panel. The image is only available to the view class itself.

⟨*View Private Draw Declaration*⟩≡
```
void drawButton(
        unsigned int button,
        bool update = true
    );
```

This method first picks which set of images to use based upon which type of button this is. Then, the method checks whether the button is on or off. It picks the appropriate image from the set accordingly. Then, it uses the rectangle defined for that button to do the blit. If it has to update the screen, it draws in the overlay for that button area, too, and then refreshes that rectangle of the screen.

⟨*View Private Draw Implementation*⟩≡

```
void
⟨ViewNameSpace⟩::drawButton(
        unsigned int button,
        bool update
    )
{
    SDL_Surface** images;

    ⟨View Draw Button pick images⟩

    if ( this->bPressed[ button ] ) {
        ++images;
    }

    ::SDL_BlitSurface(
            *images, 0,
            this->screen, &bLocation[ button ]
        );

    if ( update ) {
        ⟨View Draw Button portion of overlay⟩
        ::SDL_UpdateRect( this->screen,
                bLocation[ button ].x, bLocation[ button ].y,
                bLocation[ button ].w, bLocation[ button ].h
            );
    }
}
```

The different buttons have different images associated with them. Fortunately, they're grouped quite a bit. Which images to use is obvious from the index of the button.

⟨*View Draw Button pick images*⟩≡

```
if ( button >= DIM_2 && button <= DIM_4 ) {
    images = this->dimButton;
} else if ( button >= EASY && button <= WRAP ) {
    images = this->setButton;
} else if ( button >= NEW && button <= BACK ) {
    images = this->actButton;
} else if ( button == HELP ) {
    images = this->helpButton;
} else {
    assert( 0 == 1 );
}
```

The overlay portion that fits in this button's bounding box has slightly different coordinates than the box itself since the sidebar doesn't start at the same place the box does. Once this is taken into account, it is a simple blit to the screen.

⟨*View Draw Button portion of overlay*⟩≡

```
SDL_Rect overlayRect;
overlayRect.x = bLocation[ button ].x - SIDEBAR_X;
overlayRect.y = bLocation[ button ].y - SIDEBAR_Y;
overlayRect.w = bLocation[ button ].w;
overlayRect.h = bLocation[ button ].h;

::SDL_BlitSurface(
        this->overlay, &overlayRect,
        this->screen, &bLocation[ button ]
    );
```

6.6 The Sound Methods

There are four different sounds during the game. These are the background music, the noise made when a move happens, the victory music, and the defeat music. The `View` class defines default implementations of each of these methods.

⟨*View Sound Interface*⟩≡

```
virtual void backgroundMusic( bool stop = false );
virtual void moveNoise( void );
virtual void victoryMusic( void );
virtual void losingMusic( void );
```

At the moment, there is no background music. If time and space permit before the contest deadline, I will hook in something to play music based upon which cells are set on the board.

⟨*View Sound Implementations*⟩≡
```
void
⟨ViewNameSpace⟩::backgroundMusic( bool stop )
{
}
```

If the sound device was successfully opened, then we play a ding with each move.

⟨*View Sound Implementations*⟩+≡
```
void
⟨ViewNameSpace⟩::moveNoise( void )
{
    if ( this->sound != 0 ) {
        this->sound->ding();
    } else {
        ::SDL_Delay( 250U );
    }
}
```

At the moment, there is no victory music. If time permits, I will hook in something to play a little fanfare. For the moment, this routine simply pauses for five seconds.

⟨*View Sound Implementations*⟩+≡
```
void
⟨ViewNameSpace⟩::victoryMusic( void )
{
    ::SDL_Delay( 5000U );
}
```

At the moment, there is no losing music. If time permits, I will hook in something to play a little fanfare. For the moment, this routine simply pauses for three seconds.

⟨*View Sound Implementations*⟩+≡
```
void
⟨ViewNameSpace⟩::losingMusic( void )
{
    ::SDL_Delay( 3000U );
}
```

6.7 The Winning Method

The view class has a method which allows one to tell the user the game has been won.

⟨*View Winning Declaration*⟩≡
```
virtual void showWinning(
        unsigned int actualMoves,
        unsigned int expectedMoves
    );
```

At the moment, the actual versus expected moves here are ignored. A simple banner is displayed to congratulate the player. It is placed over the cube display area until the victory music is done.

⟨*View Winning Implementation*⟩≡
```
void
⟨ViewNameSpace⟩::showWinning(
        unsigned int /*actualMoves*/,
        unsigned int /*expectedMoves*/
    )
{
    if ( this->screen != 0 && this->victory != 0 ) {
        SDL_Rect rect;
        rect.x = ( 600 - this->victory->w ) / 2;
        rect.y = ( 600 - this->victory->h ) / 2;
        ::SDL_BlitSurface( this->victory, 0, this->screen, &rect );
        ::SDL_UpdateRect(
                this->screen,
                rect.x, rect.y,
                this->victory->w, this->victory->h
            );
    }
    this->victoryMusic();
    if ( this->screen != 0 && this->victory != 0 ) {
        this->redraw();
    }
}
```

6.8 The Losing Method

The view class has a method which allows one to tell the user the game has been lost.

⟨*View Losing Declaration*⟩≡
```
virtual void showLosing( void );
```

This method is very similar to the previous method. A simple banner is displayed to inform the player. It is placed over the cube display area until the losing music is done.

⟨*View Losing Implementation*⟩≡

```
void
⟨ViewNameSpace⟩::showLosing( void )
{
    if ( this->screen != 0 && this->losing != 0 ) {
        SDL_Rect rect;
        rect.x = ( 600 - this->losing->w ) / 2;
        rect.y = ( 600 - this->losing->h ) / 2;
        ::SDL_BlitSurface( this->losing, 0, this->screen, &rect );
        ::SDL_UpdateRect(
                this->screen,
                rect.x, rect.y,
                this->losing->w, this->losing->h
            );
    }
    this->losingMusic();
    if ( this->screen != 0 && this->losing != 0 ) {
        this->redraw();
    }
}
```

6.9 Setting the Help Mode

This method is used to allow one to set the current help context.

⟨*View Set Help Declaration*⟩≡

```
void setHelp( Help* nn );
```

This method simply releases the memory associated with any old help context and assigns the new help context from the argument.

⟨*View Set Help Implementation*⟩≡

```
void
⟨ViewNameSpace⟩::setHelp( Help* nn )
{
    delete this->help;
    this->help = nn;

    if ( this->help == 0 ) {
        this->bPressed[ HELP ] = false;
        this->drawButton( HELP );
        this->redraw();
    }
}
```

6.10 The `View` class

In this section, we assemble the `View` class from the pieces in the sections above.

The `View` class starts off by defining the constants it uses internally to size things. And, it defines the names it uses for its buttons.

⟨ *View Class Definition* ⟩≡
```
public:
    ⟨ View Constant Definitions ⟩
    ⟨ View Button Names ⟩
```

The `View` class then defines its constructor and destructor.

⟨ *View Class Definition* ⟩+≡
```
protected:
    ⟨ View Constructor Declaration ⟩
    ⟨ View Destructor Declaration ⟩
```

The `View` class then defines its reset method.

⟨ *View Class Definition* ⟩+≡
```
public:
    ⟨ View Reset Declaration ⟩
```

The `View` class defines methods to convert between screen coordinates and cell indexes.

⟨ *View Class Definition* ⟩+≡
```
public:
    ⟨ View Screen-Cell Declarations ⟩
```

The `View` class defines the interface for redrawing the view. And, it defines the functions it uses internally to draw things.

⟨ *View Class Definition* ⟩+≡
```
public:
    ⟨ View Redraw Declarations ⟩
private:
    ⟨ View Private Draw Declaration ⟩
```

The `View` class also contains a method which shows a victory or failure message on the screen. This method is declared here.

⟨ *View Class Definition* ⟩+≡
```
public:
    ⟨ View Winning Declaration ⟩
    ⟨ View Losing Declaration ⟩
```

The `View` class also defines the interface that controllers can use to play particular game sounds.

⟨ *View Class Definition* ⟩+≡
```
public:
    ⟨ View Sound Interface ⟩
```

The `View` class defines the interface for accepting mouse clicks and the method it uses internally to decide when the mouse has been clicked within a particular button.

⟨ *View Class Definition* ⟩+≡
```
public:
    ⟨ View Mouse Click Interface ⟩
private:
    ⟨ View Mouse Check Button Declaration ⟩
```

The `View` class defines a method used to set the current help context.

⟨ *View Class Definition* ⟩+≡
```
public:
    ⟨ View Set Help Declaration ⟩
```

The `View` class also declares the array it uses in its internal structures to track the starting position (in screen coordinates) of the cube display for different numbers of dimensions.

⟨ *View Class Definition* ⟩+≡
```
private:
    ⟨ View Cube Start Coordinates ⟩
```

Then, the view class declares the array that it uses to track the locations of the buttons in the sidebar.

⟨ *View Class Definition* ⟩+≡
```
public:
    ⟨ View Sidebar Location ⟩
private:
    ⟨ View Button Location ⟩
```

Then, the view class declares its instance variables.

⟨ *View Class Definition* ⟩+≡
```
protected:
    ⟨ View Screen ⟩
    ⟨ View Sound ⟩
    ⟨ View Cube ⟩
    ⟨ View Game State ⟩
private:
    ⟨ View Colors ⟩
    ⟨ View Images ⟩
    ⟨ View Button States ⟩
    ⟨ View Help ⟩
```

Once these declarations are all done, we throw all of these together into the class declaration itself. We pre-declare the `Controller` class because we need a pointer to one of them as an argument to the mouse-click handler. And, we pre-declare the `Help` class because we need a pointer to one of them as an argument to the `setHelp()` method.

⟨*View Class Declaration*⟩≡
```
class Controller;
class Help;
class View {
    ⟨View Class Definition⟩
};
```

6.11 The `view.h` file

In this section, we assemble the header file for the `View` class. It is really straightforward since we assembled the class declaration in the previous section. The only thing that we add to the class declaration is that we tuck it into our own name space so that we can keep the global namespace squeaky clean.

⟨*view.h*⟩≡
```
namespace ⟨NameSpace⟩ {
    ⟨View Class Declaration⟩
};
```

6.12 The `view.cpp` file

There is not much to the `view.cpp` source file at the moment. It contains the include files it needs for the `SDL` interactions. And, it contains the include file it needs to access the `Cube` class. It also contains the header file generated in the previous section and the header files which let it deal with the cube, the controller, the font, and the help contexts.

⟨*view.cpp*⟩≡
```
#include <assert.h>
#include <SDL.h>
#include <SDL_image.h>
#include "cube.h"
#include "soundDev.h"
#include "view.h"
#include "controller.h"
#include "font.h"
#include "help.h"
```

After that, it includes the lookup table for the starting coordinates of cubes of different dimensions. And, it includes the list of locations of its buttons.

⟨*view.cpp*⟩+≡
```
⟨View Cube Start Coordinates Definition⟩
⟨View Button Location Declaration⟩
```

Next, the implementation file includes the source code for the constructor and destructor for the `View` class.

⟨*view.cpp*⟩+≡
⟨*View Constructor Implementation*⟩
⟨*View Destructor Implementation*⟩

Next, the implementation file includes the source code for the reset method and the mouse handler and its internal code to check whether mouse events happen in particular buttons.

⟨*view.cpp*⟩+≡
⟨*View Reset Implementation*⟩
⟨*View Mouse Click Implementation*⟩
⟨*View Mouse Check Button Implementation*⟩

Then, it includes the implementation of the coordinate transformation methods.

⟨*view.cpp*⟩+≡
⟨*View Screen-Cell Implementations*⟩

After this, the source file includes the implementation of its redraw methods

⟨*view.cpp*⟩+≡
⟨*View Redraw Implementation*⟩
⟨*View Private Draw Implementation*⟩

Next, in the source file, is the default implementations of the sound methods. These are just hooks at the moment. They will be expanded if time permits.

⟨*view.cpp*⟩+≡
⟨*View Sound Implementations*⟩

Then, the code used by the view class to display the victory or loss message is included.

⟨*view.cpp*⟩+≡
⟨*View Winning Implementation*⟩
⟨*View Losing Implementation*⟩

Finally, the code used to set the current help mode is included.

⟨*view.cpp*⟩+≡
⟨*View Set Help Implementation*⟩

Part II
The Main Menu

7 The Main Menu Controller

The namespace inside the Main Menu controller class is a concatenation of the general namespace and the name of the Main Menu controller class.

⟨*MainMenuCNameSpace*⟩≡
```
⟨NameSpace⟩::MainMenuController
```

The MainMenu controller inherits from the generic controller of §5. It simply fields mouse events for the main menu screen. And, it fields clicks on the sidebar "Quit" button.

The MainMenu game controller contains an instance of the MainMenu game view.

⟨*MainMenuC View*⟩≡
```
MainMenuView view;
```

7.1 The Constructor and Destructor

The constructor for the MainMenu controller class takes two arguments. It takes a pointer to the screen as its first argument and a pointer to the game cube as the second argument. It won't actually make any use of the game cube, but it passes it into the `Controller` base class so that the base class can feel happy about life.

⟨*MainMenuC Constructor Declaration*⟩≡
```
MainMenuController(
        SDL_Surface* _screen, Cube* cube
    );
```

The constructor for the MainMenu controller class simply passes its arguments to the `Controller` constructor and the `MainMenuView` member. Then, it tells the view to redraw itself.

⟨*MainMenuC Constructor Implementation*⟩≡
```
⟨MainMenuCNameSpace⟩::MainMenuController(
        SDL_Surface* _screen,
        Cube* _cube
    ) : Controller( _cube ),
        view( _screen )
{
    this->view.redraw();
}
```

The destructor for the MainMenu controller does nothing at the moment.

⟨*MainMenuC Destructor Declaration*⟩≡
```
virtual ~MainMenuController( void );
```

⟨*MainMenuC Destructor Implementation*⟩≡

```
⟨MainMenuCNameSpace⟩::~MainMenuController( void )
{
}
```

7.2 The Mouse Event Interface

The routine which handles mouse events needs to know whether the event is a mouse press or mouse release. It also needs to know where the event happened. And, it needs to know which mouse button was pressed.

⟨*MainMenuC Mouse Click Declaration*⟩≡

```
virtual void handleMouseClick(
        bool isMouseUp,
        unsigned int xx,
        unsigned int yy,
        unsigned int buttonNumber
    );
```

In this case, it simply checks which area on the screen has been clicked. Then, it sends off the appropriate user-event to tell the main loop to reset the current controller.

⟨*MainMenuC Mouse Click Implementation*⟩≡

```
void
⟨MainMenuCNameSpace⟩::handleMouseClick(
        bool isMouseUp,
        unsigned int xx,
        unsigned int yy,
        unsigned int buttonNumber
    )
{
    unsigned int index;
    bool hit;

    hit = this->view.handleMouseClick(
                this, isMouseUp, xx, yy, buttonNumber
            );

    if ( !hit ) {
        ⟨MainMenuC Find Which Item Clicked⟩
        if ( hit ) {
            ⟨MainMenuC Fire Off Change Event⟩
        }
    }
}
```

This loop runs through each game selectable from the main menu. Then, it checks the mouse click against the bounding box of the corresponding game.

⟨*MainMenuC Find Which Item Clicked*⟩≡

```
unsigned int maxGame = ⟨MainMenuVNameSpace⟩::MAX_GAME;
unsigned int chosen;
for ( unsigned int ii=0; !hit && ii < maxGame; ++ii ) {
    if ( this->view.pointInBox( xx, yy, ii ) ) {
        chosen = ii;
        hit = true;
    }
}
```

For our purposes, the event that gets sent is a very simple event whose code completely identifies the game mode to switch into. We use `SDL_USEREVENT` to signal the game change. The code `-1` is used to switch to the main menu. After that, the games are coded in order: FlipFlop, BombSquad, MazeRunner, PegJumper, and TileSlider.

⟨*MainMenuC Fire Off Change Event*⟩≡

```
SDL_Event change;
change.type = SDL_USEREVENT;
change.user.code = chosen;
::SDL_PushEvent( &change );
```

7.3 The Game Setting Interface

The main menu isn't really a game. Therefore, it has really wimpy implementations of all of the game setting methods. They don't do anything.

⟨*MainMenuC Game Setting Interface*⟩≡

```
virtual void setDimension( unsigned int _dims );
virtual void setSkillLevel( unsigned int _skillLevel );
virtual void setWrap( bool _wrap );
virtual void newGame( void );
```

⟨*MainMenuC Game Setting Implementation*⟩≡

```
void
⟨MainMenuCNameSpace⟩::setDimension(
        unsigned int _dims
    )
{
}
```

⟨*MainMenuC Game Setting Implementation*⟩+≡
```
void
⟨MainMenuCNameSpace⟩::setSkillLevel(
        unsigned int _skillLevel
    )
{
}
```

⟨*MainMenuC Game Setting Implementation*⟩+≡
```
void
⟨MainMenuCNameSpace⟩::setWrap(
        bool _wrap
    )
{
}
```

⟨*MainMenuC Game Setting Implementation*⟩+≡
```
void
⟨MainMenuCNameSpace⟩::newGame( void )
{
}
```

7.4 The `MainMenuController` class

In this section, we assemble the `MainMenuController` class from the pieces in the sections above.

We include, in the `MainMenuController` class, the constructor and the destructor.

⟨*MainMenuC Class Definition*⟩≡
```
public:
    ⟨MainMenuC Constructor Declaration⟩
    ⟨MainMenuC Destructor Declaration⟩
```

The `MainMenuController` class also declares the methods that would be used by a more involved `View` class to change the game state.

⟨*MainMenuC Class Definition*⟩+≡
```
public:
    ⟨MainMenuC Game Setting Interface⟩
```

We include, in the `MainMenuController` class, the method used for mouse clicks.

⟨*MainMenuC Class Definition*⟩+≡
```
public:
    ⟨MainMenuC Mouse Click Declaration⟩
```

The `MainMenuController` class also contains the member variables which were defined at the beginning of this section of the document.

⟨*MainMenuC Class Definition*⟩+≡
```
private:
    ⟨MainMenuC View⟩
```

Once these declarations are all done, we throw all of these together into the class declaration itself. The `MainMenuController` inherits directly from the `Controller` class of §5 so that the main loop of the program doesn't have to do anything special for it.

⟨*MainMenuC Class Declaration*⟩≡
```
class MainMenuController : public Controller {
    ⟨MainMenuC Class Definition⟩
};
```

7.5 The `mainmenuController.h` file

In this section, we assemble the header file for the `MainMenuController` class. It is really straightforward since we assembled the class declaration in the previous section. The only thing that we add to the class declaration is that we tuck it into our own name space so that we can keep the global namespace squeaky clean.

⟨*mainmenuController.h*⟩≡
```
namespace ⟨NameSpace⟩ {
    ⟨MainMenuC Class Declaration⟩
};
```

7.6 The `mainmenuController.cpp` file

In this section, we assemble the MainMenu controller source file. It requires the header files for the `Cube` class, the `SoundDev` class, the `View` class, the `Controller` class, and the `MainMenuView` classes in addition to its own header file.

⟨*mainmenuController.cpp*⟩≡
```
#include <SDL.h>
#include "cube.h"
#include "soundDev.h"
#include "view.h"
#include "controller.h"
#include "mainmenuView.h"
#include "mainmenuController.h"
```

After the header files, we include the implementations of the constructor and destructor.

⟨*mainmenuController.cpp*⟩+≡
⟨*MainMenuC Constructor Implementation*⟩
⟨*MainMenuC Destructor Implementation*⟩

After the constructor and destructor, the implementations of the bogus game state methods are also included.

⟨*mainmenuController.cpp*⟩+≡
⟨*MainMenuC Game Setting Implementation*⟩

Then, we include the implementation of the method used to field mouse clicks.

⟨*mainmenuController.cpp*⟩+≡
⟨*MainMenuC Mouse Click Implementation*⟩

8 The Main Menu View

The namespace inside the MainMenu view class is a concatenation of the general namespace and the name of the MainMenu view class.

⟨*MainMenuVNameSpace*⟩≡
⟨*NameSpace*⟩::MainMenuView

The main menu defines all of the games selectable from the main menu. This ordering is used in the arrays of images and bounding boxes below. The ordering is also used by the main menu controller to tell the main loop when to switch to a different game mode.

⟨*MainMenu Games*⟩≡
```
enum {
    FLIPFLOP = 0,
    BOMBSQUAD,
    MAZERUNNER,
    PEGJUMPER,
    TILESLIDER,
    MAX_GAME
};
```

The Main Menu stores pointers to the images of the sidebar, the backdrop, and the button overlay.

⟨*MainMenu Images*⟩≡
```
SDL_Surface* sidebar;
SDL_Surface* backdrop;
SDL_Surface* overlay;
```

The Main Menu stores pointers to the images of the game logos to use.

⟨*MainMenu Images*⟩+≡
```
SDL_Surface* logos[ MAX_GAME ];
```

The Main Menu view class also stores the images used as a backdrop for the quit button.

⟨*MainMenu Images*⟩+≡
```
SDL_Surface* quitButton[2];
```

In addition to that, it also keeps track of where the quit button is on the screen and whether it is pressed or not.

⟨*MainMenu Box Declarations*⟩≡
```
static SDL_Rect quitBox;
```

⟨*MainMenu Box Definitions*⟩≡
```
SDL_Rect ⟨MainMenuVNameSpace⟩::quitBox = {
    ⟨ViewNameSpace⟩::SIDEBAR_X + 2,
    ⟨ViewNameSpace⟩::SIDEBAR_Y + 534,
    196, 64
};
```

⟨*MainMenu Quit State*⟩≡
```
bool quitPressed;
```

The Main Menu also caches a pointer to the screen.

⟨*MainMenu Screen*⟩≡
```
SDL_Surface* screen;
```

The Main Menu also tracks the bounding boxes of each of the above logos.

⟨*MainMenu Box Declarations*⟩+≡
```
static SDL_Rect boxes[ MAX_GAME ];
```

⟨*MainMenu Box Definitions*⟩+≡
```
SDL_Rect ⟨MainMenuVNameSpace⟩::boxes[ MAX_GAME ] = {
    {   0,   0, 300, 200 },
    { 300,   0, 300, 200 },
    { 150, 200, 300, 200 },
    {   0, 400, 300, 200 },
    { 300, 400, 300, 200 },
};
```

8.1 The Constructor

The constructor for the MainMenu view class takes one argument—a pointer to the screen.

⟨*MainMenuV Constructor Declaration*⟩≡
```
MainMenuView( SDL_Surface* _screen );
```

The constructor for the MainMenu view class loads the images for the sidebar and quit button and for each of the games.

⟨*MainMenuV Constructor Implementation*⟩≡
```
⟨MainMenuVNameSpace⟩::MainMenuView(
        SDL_Surface* _screen
    ) : quitPressed( false ), screen( _screen )
{
    ⟨MainMenuV Load Base Images⟩
    ⟨MainMenuV Load Logo Images⟩
}
```

There are images for the backdrop of the sidebar, the backdrop of the game, the overlay over the sidebar buttons, and the quit button in the up and down states.

⟨*MainMenuV Load Base Images*⟩≡
```
this->sidebar = ::IMG_Load( "../../data/panel.png" );
this->backdrop = ::IMG_Load( "../../data/backdrop.png" );
this->overlay = ::IMG_Load( "../../data/moverlay.png" );
this->quitButton[ 0 ] = ::IMG_Load( "../../data/helpOff.png" );
this->quitButton[ 1 ] = ::IMG_Load( "../../data/helpOn.png" );
```

There are also logos for each of the games available from the main menu.

⟨*MainMenuV Load Logo Images*⟩≡
```
this->logos[ FLIPFLOP ]
    = ::IMG_Load( "../../data/flogo.png" );
this->logos[ BOMBSQUAD ]
    = ::IMG_Load( "../../data/blogo.png" );
this->logos[ MAZERUNNER ]
    = ::IMG_Load( "../../data/mlogo.png" );
this->logos[ PEGJUMPER ]
    = ::IMG_Load( "../../data/plogo.png" );
this->logos[ TILESLIDER ]
    = ::IMG_Load( "../../data/tlogo.png" );
```

8.2 The Destructor

The destructor for the MainMenu view class simply release the images loaded above in the constructor.

⟨*MainMenuV Destructor Declaration*⟩≡
```
~MainMenuView( void );
```

⟨*MainMenuV Destructor Implementation*⟩≡
```
⟨MainMenuVNameSpace⟩::~MainMenuView( void )
{
    ⟨MainMenuV Release Logo Images⟩
    ⟨MainMenuV Release Base Images⟩
}
```

Releasing the logo images is easy. We can just loop through each of the logos and release them.

⟨*MainMenuV Release Logo Images*⟩≡
```
for ( unsigned int ii=0; ii < MAX_GAME; ++ii ) {
    ::SDL_FreeSurface( this->logos[ ii ] );
}
```

Releasing the base images takes a little more effort. We have to release each image one by one.

⟨*MainMenuV Release Base Images*⟩≡
```
::SDL_FreeSurface( this->quitButton[ 1 ] );
::SDL_FreeSurface( this->quitButton[ 0 ] );
::SDL_FreeSurface( this->overlay );
::SDL_FreeSurface( this->backdrop );
::SDL_FreeSurface( this->sidebar );
```

8.3 The Redraw Methods

The MainMenu view class has a method which allows one to update the entire display area for the game.

⟨*MainMenuV Redraw Declarations*⟩≡
```
void redraw( void ) const;
```

The redraw function here simply redraws the backdrop and the sidebar and each of the game logos.

⟨*MainMenuV Redraw Implementations*⟩≡
```
void
⟨MainMenuVNameSpace⟩::redraw( void ) const
{
    SDL_Rect rr;
    ⟨MainMenu Redraw Backdrop⟩
    ⟨MainMenu Redraw Logos⟩
    ⟨MainMenu Redraw Sidebar⟩
    ::SDL_UpdateRect( this->screen, 0, 0, 0, 0 );
}
```

Redrawing the backdrop is simple. We simply have to copy the backdrop onto the screen.

⟨*MainMenu Redraw Backdrop*⟩≡
```
::SDL_BlitSurface( this->backdrop, 0, this->screen, 0 );
```

To draw the logos for the games onto the screen, we have to copy each of them into their appropriate rectangle.

⟨*MainMenu Redraw Logos*⟩≡
```
for ( unsigned int ii=0; ii < MAX_GAME; ++ii ) {
    rr = boxes[ ii ];
    ::SDL_BlitSurface( this->logos[ ii ], 0, this->screen, &rr );
}
```

To draw the sidebar, we first draw the backdrop of the sidebar.

⟨*MainMenu Redraw Sidebar*⟩≡
```
rr.x = ⟨ViewNameSpace⟩::SIDEBAR_X;
rr.y = ⟨ViewNameSpace⟩::SIDEBAR_Y;
::SDL_BlitSurface( this->sidebar, 0, this->screen, &rr );
```

After that, we draw the overlay atop the sidebar.

⟨*MainMenu Redraw Sidebar*⟩+≡
```
rr.x = ⟨ViewNameSpace⟩::SIDEBAR_X;
rr.y = ⟨ViewNameSpace⟩::SIDEBAR_Y;
::SDL_BlitSurface( this->overlay, 0, this->screen, &rr );
```

Then, we redraw the quit button on top of the sidebar.

⟨*MainMenu Redraw Sidebar*⟩+≡
```
this->redrawQuit( false );
```

The above draw method and the handler for mouse events both need to draw the quit button. Rather than duplicating code, we have broken it out into its own method.

⟨*MainMenuV Redraw Declarations*⟩+≡

```
void redrawQuit( bool refresh = true ) const;
```

The method simply picks the appropriate bitmap for the current state of the button. Then, it draws the button. Then, it copies the portion of the overlay that would otherwise have been atop the button. Then, if it is supposed to refresh, it updates the appropriate rectangle of the screen.

⟨*MainMenuV Redraw Implementations*⟩+≡

```
void
⟨MainMenuVNameSpace⟩::redrawQuit( bool refresh ) const
{
    unsigned int index = ( ! this->quitPressed ) ? 0 : 1 ;
    SDL_Surface* button = this->quitButton[ index ];
    ::SDL_BlitSurface( button, 0, this->screen, &quitBox );

    ⟨MainMenuV Redraw Quit Show Overlay⟩

    if ( refresh ) {
        ::SDL_UpdateRect(
                this->screen,
                quitBox.x, quitBox.y,
                quitBox.w, quitBox.h
            );
    }
}
```

Because the overlay is the same size as the sidebar, we have to take extra care to make sure that we're copying the right portion of it onto the button.

⟨*MainMenuV Redraw Quit Show Overlay*⟩≡

```
SDL_Rect rr = quitBox;
rr.x -= ⟨ViewNameSpace⟩::SIDEBAR_X;
rr.y -= ⟨ViewNameSpace⟩::SIDEBAR_Y;
::SDL_BlitSurface( this->overlay, &rr, this->screen, &quitBox );
```

8.4 Handling Mouse Clicks

When the Main Menu controller receives a mouse click, it passes it on to the `MainMenuView` class. The view class is responsible for determining if the quit button was clicked. If it was, then the quit event is sent out.

⟨*MainMenuV Mouse Click Declaration*⟩≡

```
virtual bool handleMouseClick(
        Controller* controller,
        bool isMouseUp,
        unsigned int xx,
        unsigned int yy,
        unsigned int buttonNumber
    );
```

This method checks to see if the event happened inside the quit button. If we hit the quit button with a mouse down event, then we put the quit button into the "pressed" state. Otherwise, if the quit button had already been pressed, we put it back into the "unpressed" state if the event is a mouse up event and we actually quit if the mouse up event happened inside of the pressed quit button.

⟨*MainMenuV Mouse Click Implementation*⟩≡

```
bool
⟨MainMenuVNameSpace⟩::handleMouseClick(
        Controller* control,
        bool isMouseUp,
        unsigned int xx,
        unsigned int yy,
        unsigned int buttonNumber
    )
{
    ⟨MainMenuV MouseClick Check Quit Button⟩

    if ( hitQuit && ! isMouseUp ) {
        ⟨MainMenuV MouseClick Press Quit⟩
    } else if ( this->quitPressed )  {
        if ( isMouseUp ) {
            ⟨MainMenuV MouseClick Unpress Quit⟩
        }
        if ( hitQuit ) {
            ⟨MainMenuV MouseClick Perform Quit⟩
        }
    }

    return hitQuit;
}
```

Checking to see if the mouse event happened inside the quit button is a straightforward check of the coordinates with the bounds of the quit button.

⟨*MainMenuV MouseClick Check Quit Button*⟩≡
```
bool hitQuit = (
           xx >= quitBox.x && xx < quitBox.x + quitBox.w
        && yy >= quitBox.y && yy < quitBox.y + quitBox.h
    );
```

To set the button into the pressed mode, we mark it as pressed and redraw it.

⟨*MainMenuV MouseClick Press Quit*⟩≡
```
this->quitPressed = true;
this->redrawQuit();
```

To set the button into the unpressed mode, we mark it as unpressed and redraw it.

⟨*MainMenuV MouseClick Unpress Quit*⟩≡
```
this->quitPressed = false;
this->redrawQuit();
```

If someone pressed and then released the mouse within the quit button, then we actually send off the SDL event to signal a quit.

⟨*MainMenuV MouseClick Perform Quit*⟩≡
```
SDL_Event quit;
quit.type = SDL_QUIT;
::SDL_PushEvent( &quit );
```

8.5 The Point in Box Method

The controller uses this method to see if the given point is inside a particular game's bounding box.

⟨*MainMenu Point In Box Declaration*⟩≡
```
bool pointInBox(
        unsigned int xx, unsigned int yy,
        unsigned int game
    ) const;
```

The method itself simply compares the coordinates with those of the box in question.

⟨*MainMenu Point In Box Implementation*⟩≡

```
bool
⟨MainMenuVNameSpace⟩::pointInBox(
        unsigned int xx, unsigned int yy,
        unsigned int game
    ) const
{
    assert( game < MAX_GAME );
    return xx >= boxes[ game ].x
        && xx <  boxes[ game ].x + boxes[ game ].w
        && yy >= boxes[ game ].y
        && yy <  boxes[ game ].y + boxes[ game ].h;
}
```

8.6 The `MainMenuView` class

In this section, we assemble the `MainMenuView` class from the pieces in the sections above.

We start off the `MainMenuView` class with the declaration of which games are supported.

⟨*MainMenuV Class Definition*⟩≡

```
public:
    ⟨MainMenu Games⟩
```

We include, in the `MainMenuView` class, the constructor, the destructor, and the redraw methods.

⟨*MainMenuV Class Definition*⟩+≡

```
public:
    ⟨MainMenuV Constructor Declaration⟩
    ⟨MainMenuV Destructor Declaration⟩
    ⟨MainMenuV Redraw Declarations⟩
```

The `MainMenuView` class then includes the declaration of the method used to see if a particular button was clicked and the declaration of the method used to field clicks on the quit button.

⟨*MainMenuV Class Definition*⟩+≡

```
public:
    ⟨MainMenu Point In Box Declaration⟩
    ⟨MainMenuV Mouse Click Declaration⟩
```

We include the variables that are used in the main menu view class.

⟨*MainMenuV Class Definition*⟩+≡

```
private:
    ⟨MainMenu Images⟩
    ⟨MainMenu Screen⟩
    ⟨MainMenu Quit State⟩
    ⟨MainMenu Box Declarations⟩
```

Once these declarations are all done, we throw all of these together into the class declaration itself.

⟨*MainMenuV Class Declaration*⟩≡

```
class MainMenuView {
    ⟨MainMenuV Class Definition⟩
};
```

8.7 The `mainmenuView.h` file

In this section, we assemble the header file for the `MainMenuView` class. It is really straightforward since we assembled the class declaration in the previous section. The only thing that we add to the class declaration is that we tuck it into our own name space so that we can keep the global namespace squeaky clean.

⟨*mainmenuView.h*⟩≡

```
namespace ⟨NameSpace⟩ {
    ⟨MainMenuV Class Declaration⟩
};
```

8.8 The `mainmenuView.cpp` file

In this section, we assemble the MainMenu view source file. It requires the `SDL` headers for dealing with surfaces, the screen, blitting, and loading images. It requires the header files for the `Cube` class, the `SoundDev` class, the `View` class for its constants, and the `MainMenuView` class itself.

⟨*mainmenuView.cpp*⟩≡

```
#include <assert.h>
#include <SDL.h>
#include <SDL_image.h>
#include "cube.h"
#include "soundDev.h"
#include "view.h"
#include "mainmenuView.h"
```

After the header files, we include the implementations of the constructor, the destructor, and the redraw methods.

⟨*mainmenuView.cpp*⟩+≡
⟨*MainMenuV Constructor Implementation*⟩
⟨*MainMenuV Destructor Implementation*⟩
⟨*MainMenuV Redraw Implementations*⟩

The main menu source file then goes on to include the implementation of the method used to check to see if a particular box has been clicked and the implementation of the method used to check to see if the quit button has been clicked.

⟨*mainmenuView.cpp*⟩+≡
⟨*MainMenu Point In Box Implementation*⟩
⟨*MainMenuV Mouse Click Implementation*⟩

Then, the source file includes the definition of the bounding boxes.

⟨*mainmenuView.cpp*⟩+≡
⟨*MainMenu Box Definitions*⟩

9 The Help Screen Class

The namespace inside the help screen class is a concatenation of the general namespace and the name of the help screen class.

⟨*HelpNameSpace*⟩≡
```
⟨NameSpace⟩::Help
```

The `Help` class contains a pointer to the current view so that it can update the help class.

⟨*Help View*⟩≡
```
View* view;
```

The `Help` class contains a pointer to the screen for output

⟨*Help Screen*⟩≡
```
enum { X_OFFSET = 20 };
enum { Y_OFFSET = 20 };
SDL_Surface* screen;
```

The `Help` class keeps a copy of the font that it uses to display the actual text of the help messages.

⟨*Help Font*⟩≡
```
Font* font;
```

The `Help` class also keeps an array of all of the hot spots that are currently active.

⟨*Help Hot Spots*⟩≡
```
enum { MAX_HOTSPOT = 32 };
unsigned int hotSpotCount;
struct {
    char fname[ 128 ];
    unsigned int x;
    unsigned int y;
    unsigned int w;
    unsigned int h;
} hotSpots[ MAX_HOTSPOT ];
unsigned int clickedHotSpot;
```

9.1 The Constructor

The constructor for the help screen class takes three arguments. The first is a pointer to the view, the second is a pointer to the screen, and the third is an optional name of the help-context file to open.

⟨*Help Constructor Declaration*⟩≡
```
Help(
        View* _view,
        SDL_Surface* _screen,
        const char* fname = "top"
    );
```

The constructor copies the view and screen into local variables. Then, it loads the font. After that, it clears the screen and attempts to load the specified filename.

⟨*Help Constructor Implementation*⟩≡

```
⟨HelpNameSpace⟩::Help(
        View* _view,
        SDL_Surface* _screen,
        const char* fname
    ) : view( _view ), screen( _screen ), hotSpotCount( 0 )
{
    this->font = new Font();

    SDL_Rect rect;
    rect.x = 0;
    rect.y = 0;
    rect.w = 600;
    rect.h = 600;

    ::SDL_FillRect(
            this->screen,
            &rect,
            ::SDL_MapRGB( this->screen->format, 0, 0, 0 )
        );
    ::SDL_UpdateRect( this->screen, 0, 0, 0, 0 );

    this->load( fname );
}
```

The destructor for the help screen must release the font loaded above.

⟨*Help Destructor Declaration*⟩≡

```
virtual ~Help( void );
```

⟨*Help Destructor Implementation*⟩≡

```
⟨HelpNameSpace⟩::~Help( void )
{
    delete this->font;
}
```

9.2 Handling Mouse Clicks

When the view class receives a mouse click, it passes it on to the `Help` class. The help class is responsible for determining if any of its hot spots were clicked. If they were, then the appropriate file should be loaded.

⟨*Help Mouse Click Declaration*⟩≡

```
virtual bool handleMouseClick(
        bool isMouseUp,
        unsigned int xx,
        unsigned int yy,
        unsigned int buttonNumber
    );
```

If the event is a "MouseDown" event, this method loops through each hot spot to see which hot spot (if any) was clicked. If a hot spot was clicked, this is stored in the `clickedHotSpot` member. On a mouse up event where the `clickedHotSpot` member had been set, the button is set back to its original state. Then, if the release was still inside the button, the appropriate action for that button takes place. This method returns `true` if the click was anywhere inside of the appropriate area for help clicks.

⟨*Help Mouse Click Implementation*⟩≡

```
bool
⟨HelpNameSpace⟩::handleMouseClick(
        bool isMouseUp,
        unsigned int xx,
        unsigned int yy,
        unsigned int buttonNumber
    )
{
    if ( ! isMouseUp ) {
        this->clickedHotSpot = MAX_HOTSPOT;
        for ( unsigned int ii=0; ii < this->hotSpotCount; ++ii ) {
            if ( this->checkHotSpot( xx, yy, ii ) ) {
                this->clickedHotSpot = ii;
                break;
            }
        }
    } else if ( isMouseUp && this->clickedHotSpot < MAX_HOTSPOT ) {
        unsigned int ii = this->clickedHotSpot;
        bool inSide = this->checkHotSpot( xx, yy, ii );

        if ( inSide ) {
            this->load( this->hotSpots[ ii ].fname );
        }
        this->clickedHotSpot = MAX_HOTSPOT;
    }

    return ( xx < 600 && yy < 600 );
}
```

The mouse click function often needs to check whether a point is inside the bounding box for a particular hot spot. This method does that so that the same code doesn't have to be compiled twice.

⟨*Help Mouse Check Hot Spot Declaration*⟩≡

```
bool checkHotSpot(
        unsigned int xx, unsigned int yy,
        unsigned int ii
    );
```

The implementation is straightforward. It simply checks to make sure that both the x- and y-coordinates are within the box.

⟨*Help Mouse Check Hot Spot Implementation*⟩≡

```
bool
⟨HelpNameSpace⟩::checkHotSpot(
        unsigned int xx, unsigned int yy,
        unsigned int ii
    )
{
    return ( xx >= this->hotSpots[ ii ].x
        && xx < this->hotSpots[ ii ].x + this->hotSpots[ ii ].w
        && yy >= this->hotSpots[ ii ].y
        && yy < this->hotSpots[ ii ].y + this->hotSpots[ ii ].h
    );
}
```

9.3 Loading the Help File

This method is used to allow one to load a particular help context.

⟨*Help Load Declaration*⟩≡

```
void load( const char* baseName );
```

This method first resets the counter on the number of hot spots back to zero. Then, it attempts to parse the file given by `fname`. If there are any errors in that process, it bails and resets the help context to null.

⟨*Help Load Implementation*⟩≡

```
void
⟨HelpNameSpace⟩::load( const char* baseName )
{
    this->hotSpotCount = 0;

    char fname[ 512 ];
    sprintf( fname, "../../data/%s.hlp", baseName );

    ⟨Help Load parse file⟩
    if ( err ) {
        this->view->setHelp( 0 );
    }
}
```

We open the file and read a line at a time. We process each line on its own.

⟨*Help Load parse file*⟩≡
```
FILE* fp = fopen( fname, "r" );
bool err = ( fp == 0 );
char buf[ 1024 ];

while ( ! err && fgets( buf, 1024, fp ) != 0 ) {
    ⟨Help Load parse line⟩
}

if ( fp != 0 ) {
    fclose( fp );
}
```

Each of the available commands has its own subsection below. This section just looks for the beginning string of the message.

⟨*Help Load parse line*⟩≡
```
if ( strncmp( buf, "rect ", 5 ) == 0 ) {
    ⟨Help Load handle rect⟩
} else if ( strncmp( buf, "text_center ", 12 ) == 0 ) {
    ⟨Help Load handle text-center⟩
} else if ( strncmp( buf, "image ", 6 ) == 0 ) {
    ⟨Help Load handle image⟩
} else if ( strncmp( buf, "subimage ", 9 ) == 0 ) {
    ⟨Help Load handle subimage⟩
} else if ( strncmp( buf, "button ", 7 ) == 0 ) {
    ⟨Help Load handle button⟩
} else if ( strncmp( buf, "update ", 7 ) == 0 ) {
    ⟨Help Load handle update⟩
}
```

The rectangles consist of an x, y, width, and height followed by the red, green, and blue color components. The x and y coordinates are adjusted by the offsets of the help window.

⟨*Help Load handle rect*⟩≡

```
int xx;
int yy;
unsigned int ww;
unsigned int hh;
unsigned int rr;
unsigned int gg;
unsigned int bb;

sscanf( &buf[ 5 ], "%d %d %u %u %u %u %u",
        &xx, &yy, &ww, &hh,
        &rr, &gg, &bb
    );
xx += X_OFFSET;
yy += Y_OFFSET;

SDL_Rect rect;
rect.x = xx;
rect.y = yy;
rect.w = ww;
rect.h = hh;

::SDL_FillRect(
        this->screen,
        &rect,
        ::SDL_MapRGB( this->screen->format, rr, gg, bb )
    );
```

To center text, we simply read the x and y location, then we offset the position by the offsets of the help screen area. Then, we blit the message using the appropriate method on the font class.

⟨*Help Load handle text-center*⟩≡

```
unsigned int xx;
unsigned int yy;

sscanf( &buf[ 12 ], "%u %u", &xx, &yy );

xx += X_OFFSET;
yy += Y_OFFSET;

char* ptr = strrchr( buf, '"' );
*ptr = '\0';

ptr = strchr( buf, '"' );
if ( ptr != 0 ) {
    this->font->centerMessage(
            this->screen, false,
            xx, yy,
            "%s", &ptr[ 1 ]
        );
}
```

To display an image, we read in the x and y coordinates of the image. We offset those coordinates by the offset of the help viewing area. Then, we load the image, blit it, and release it.

⟨*Help Load handle image*⟩≡

```
char base[ 256 ];
unsigned int xx;
unsigned int yy;

sscanf( &buf[ 6 ], "%s %u %u", base, &xx, &yy );
xx += X_OFFSET;
yy += Y_OFFSET;

sprintf( buf, "../../data/%s.png", base );
SDL_Surface* img = ::IMG_Load( buf );

if ( img != 0 ) {
    SDL_Rect rect;
    rect.x = xx;
    rect.y = yy;

    ::SDL_BlitSurface( img, 0, this->screen, &rect );
    ::SDL_FreeSurface( img );
}
```

To display a subimage, we read in the x and y coordinates of the image and the portion of the image to blit. We offset those coordinates by the offset of the help viewing area. Then, we load the image, blit it, and release it.

⟨*Help Load handle subimage*⟩≡

```
char base[ 256 ];
unsigned int xx;
unsigned int yy;
unsigned int sx;
unsigned int sy;
unsigned int sw;
unsigned int sh;

sscanf( &buf[ 9 ], "%s %u %u %d %d %u %u",
        base, &xx, &yy,
        &sx, &sy, &sw, &sh
    );

xx += X_OFFSET;
yy += Y_OFFSET;

sprintf( buf, "../../data/%s.png", base );
SDL_Surface* img = ::IMG_Load( buf );

if ( img != 0 ) {
    SDL_Rect src;
    src.x = sx;
    src.y = sy;
    src.w = sw;
    src.h = sh;

    SDL_Rect rect;
    rect.x = xx;
    rect.y = yy;
    rect.w = src.w;
    rect.h = src.h;

    ::SDL_BlitSurface( img, &src, this->screen, &rect );
    ::SDL_FreeSurface( img );
}
```

And, to handle a hotspot, we record the basename of the file it refers to and the bounds of the hotspot. This is inherently unsafe. The "`%s`" in the `sscanf()` call could easily overflow the `128` characters we allocated for `fname`. But, since I am creating the help files, too, I'm not going to make this code overly robust at the moment.

⟨*Help Load handle button*⟩≡
```
sscanf( &buf[ 7 ], "%s %u %u %u %u",
        this->hotSpots[ this->hotSpotCount ].fname,
        &this->hotSpots[ this->hotSpotCount ].x,
        &this->hotSpots[ this->hotSpotCount ].y,
        &this->hotSpots[ this->hotSpotCount ].w,
        &this->hotSpots[ this->hotSpotCount ].h
    );
this->hotSpots[ this->hotSpotCount ].x += X_OFFSET;
this->hotSpots[ this->hotSpotCount ].y += Y_OFFSET;
++this->hotSpotCount;
```

To handle an update, we read in the coordinates to update and we refresh them.

⟨*Help Load handle update*⟩≡
```
unsigned int xx;
unsigned int yy;
unsigned int ww;
unsigned int hh;

sscanf( &buf[ 7 ], "%u %u %u %u", &xx, &yy, &ww, &hh );
xx += X_OFFSET;
yy += Y_OFFSET;

::SDL_UpdateRect( this->screen, xx, yy, ww, hh );
```

9.4 The `Help` class

In this section, we assemble the `Help` class from the pieces in the sections above.

The `Help` class starts off by defining its constructor and destructor.

⟨*Help Class Definition*⟩≡
```
public:
    ⟨Help Constructor Declaration⟩
    ⟨Help Destructor Declaration⟩
```

The `Help` class then defines the method it uses to field mouse clicks from the `View` class and the method it uses internally to see if a mouse click is inside a hot spot.

⟨*Help Class Definition*⟩+≡
```
public:
    ⟨Help Mouse Click Declaration⟩
private:
    ⟨Help Mouse Check Hot Spot Declaration⟩
```

The `Help` class then defines the method it uses to load a help file.

⟨*Help Class Definition*⟩+≡
```
private:
    ⟨Help Load Declaration⟩
```

Then, the view class declares its instance variables.

⟨*Help Class Definition*⟩+≡
```
private:
    ⟨Help View⟩
    ⟨Help Screen⟩
    ⟨Help Font⟩
    ⟨Help Hot Spots⟩
```

Once these declarations are all done, we throw all of these together into the class declaration itself.

⟨*Help Class Declaration*⟩≡
```
class Help {
    ⟨Help Class Definition⟩
};
```

9.5 The `help.h` file

In this section, we assemble the header file for the `Help` class. It is really straightforward since we assembled the class declaration in the previous section. The only thing that we add to the class declaration is that we tuck it into our own name space so that we can keep the global namespace squeaky clean. And, we pre-declare the `View` class so that we don't get into circular header dependencies.

⟨*help.h*⟩≡
```
namespace ⟨NameSpace⟩ {
    class View;
    ⟨Help Class Declaration⟩
};
```

9.6 The `help.cpp` file

There is not much to the `help.cpp` source file at the moment. It contains the include files it needs for the `SDL` interactions. It also contains the header file generated in the previous section and the header files which let it use the `Font` class and the `View` class.

⟨*help.cpp*⟩≡

```
#include <assert.h>
#include <SDL.h>
#include <SDL_image.h>
#include <string.h>
#include "cube.h"
#include "soundDev.h"
#include "view.h"
#include "font.h"
#include "help.h"
```

The implementation file includes the source code for the constructor and destructor for the `Help` class.

⟨*help.cpp*⟩+≡

⟨*Help Constructor Implementation*⟩
⟨*Help Destructor Implementation*⟩

Next, the implementation file includes the source code for the mouse click handler and the method it uses internally to tell if a hot spot was pressed.

⟨*help.cpp*⟩+≡

⟨*Help Mouse Click Implementation*⟩
⟨*Help Mouse Check Hot Spot Implementation*⟩

Finally, the implementation file includes the source code for the method which loads new help files.

⟨*help.cpp*⟩+≡

⟨*Help Load Implementation*⟩

Part III
The Flip Flop Game

10 Flip Flop

The namespace inside the flip-flop class is a concatenation of the general namespace and the name of the flip-flop class.

⟨*FlipFlopNameSpace*⟩≡
  ⟨*NameSpace*⟩`::FlipFlop`

The `FlipFlop` class keeps a pointer to the cube used for the game.

⟨*FlipFlop Cube*⟩≡
```
Cube* cube;
```

The `FlipFlop` class also tracks the number of dimensions that are being used. It needs this information so that it can properly determine the neighbors of a given point.

⟨*FlipFlop Dimensions*⟩≡
```
unsigned int dims;
```

And, the `FlipFlop` class tracks the current skill level.

⟨*FlipFlop Skill Level*⟩≡
```
unsigned int skillLevel;
```

The `FlipFlop` class also keeps track of whether or not it is wrapping around. This is necessary so that it can properly determine neighbors of things near the edge.

⟨*FlipFlop Wrap*⟩≡
```
bool wrap;
```

The `FlipFlop` class also keeps track of the number of cells which are currently on. This is used to track the winning condition.

⟨*FlipFlop On Count*⟩≡
```
int onCount;
```

Also, to track the winning condition, there is a flag that tells whether the game has already been won or not. After a person wins, she can play around on the board as much as she likes before hitting the "New" button.

⟨*FlipFlop Has Won*⟩≡
```
bool hasWon;
```

In addition, the game also tracks how many moves were expected and how many moves were taken. These can be used to give feedback to the player. At the moment, these are not used at all.

⟨*FlipFlop Move Counters*⟩≡
```
unsigned int actualMoves;
unsigned int expectedMoves;
```

The `FlipFlop` class also keeps track of the view pointer.

⟨*FlipFlop View*⟩≡
```
FlipFlopView* view;
```

10.1 The Constructor

The constructor for the `FlipFlop` class takes five arguments. The first is a pointer to the game cube, the second specifies the number of dimensions to employ, the third specifies the skill level to use, the fourth specifies whether the edges wrap around, and the fifth is an optional pointer to the view to update when cells change.

⟨*FlipFlop Constructor Declaration*⟩≡
```
FlipFlop(
        Cube* _cube,
        unsigned int _dims = 2,
        unsigned int _skillLevel = 0,
        bool _wrap = true,
        FlipFlopView* _view = 0
    );
```

The constructor for the `FlipFlop` class copies the arguments into its local variables. Then, it calls its own reset method to start a new game. But, first, it verifies that all of the input arguments match its range expectations.

⟨*FlipFlop Constructor Implementation*⟩≡
```
⟨FlipFlopNameSpace⟩::FlipFlop(
        Cube* _cube,
        unsigned int _dims,
        unsigned int _skillLevel,
        bool _wrap,
        FlipFlopView* _view
    ) : cube( _cube ),
        dims( _dims ),
        skillLevel( _skillLevel ),
        wrap( _wrap ),
        view( _view )
{
    assert( cube != 0 );
    assert( dims >= 1 );
    assert( dims <= ⟨CubeNameSpace⟩::DIMENSIONS );
    assert( skillLevel < 3 );
    this->reset();
}
```

10.2 The Reset Method

This method is used to start a new game. It requires no parameters. It assumes that both the number of dimensions and the wrap mode have already been set.

⟨*FlipFlop Reset Declaration*⟩≡
```
void reset( void );
```

The cube is cleared. Then, the skill level is used to determine the number of cells to toggle. Then, the toggles are performed, the statistics are reset, and the view is refreshed.

⟨*FlipFlop Reset Implementation*⟩≡
```
void
⟨FlipFlopNameSpace⟩::reset( void )
{
    *this->cube = 0;
    this->onCount = 0;

    ⟨FlipFlop ToggleTable⟩
    unsigned int toggles = table[ this->dims ][ this->skillLevel ];
    unsigned int len
        = ⟨CubeNameSpace⟩::arrayLengths[ this->dims ];

    ⟨FlipFlop Perform Flips⟩
    ⟨FlipFlop Reset Current Statistics⟩

    if ( this->view != 0 ) {
        this->view->reset();
        this->view->redraw();
    }
}
```

The following table is used to determine the number of cells to toggle based upon the number of dimensions and the skill level.

⟨*FlipFlop ToggleTable*⟩≡
```
unsigned int table[ Cube::DIMENSIONS+1 ][ 3 ] = {
    {  0,  0,  0 },
    {  1,  2,  3 },
    {  3,  5,  8 },
    {  5,  8, 12 },
    {  8, 16, 32 },
};
```

We want to be careful not to flip the same cell twice. So, we make a lookup table of cells we've already flipped. Then, we go through and perform each flip.

⟨*FlipFlop Perform Flips*⟩≡
```
unsigned int* lut = new unsigned int[ toggles ];
unsigned int lutLen = 0;

for ( unsigned int ii=0; ii < toggles; ++ii ) {
    ⟨FlipFlop Perform Single Flip⟩
}

delete[] lut;
```

Each time we want to flip, we pick an index from the available range. Then, we loop through the lookup table incrementing index each time we find a number that is less than or equal to index. This makes index be the `index`-th number which has not yet been chosen. Then, we perform the flip and add index into the lookup table.

⟨*FlipFlop Perform Single Flip*⟩≡
```
unsigned int index = random() % len--;
for ( unsigned int jj=0; jj < lutLen; ++jj ) {
    if ( index >= lut[ jj ] ) {
        ++index;
    }
}

this->flip( index, false );

⟨FlipFlop Add Index Into Flip LUT⟩
```

Here, we put it at the end of the lookup table. Then, we keep trying to swap it with the element before it in the list until we come to an element before it that is less than it. We need to keep the lookup table sorted for the previous incrementing loop to work right. This is just a simple bit of a bubble sort where we know that the only element potentially out of place is the last one.

⟨*FlipFlop Add Index Into Flip LUT*⟩≡
```
unsigned int spot = lutLen;
while ( spot > 0 && lut[ spot-1 ] > index ) {
    lut[ spot ] = lut[ spot-1 ];
    --spot;
}
lut[ spot ] = index;
++lutLen;
```

Resetting the statistics for the current game is easy. It is expected that the number of moves that it will take to solve the puzzle is the same as the number of toggles. However, it could quite possibly take fewer moves, so we shouldn't rely upon this metric.

⟨*FlipFlop Reset Current Statistics*⟩≡
```
this->hasWon = false;
this->actualMoves = 0;
this->expectedMoves = toggles;
```

10.3 The Flip Method

The flip method will be used in the `reset()` method above and by the flip-flop controller to register moves. It will simply toggle the cube entry at `index` and at each of `index`'s neighbors.

⟨*FlipFlop Flip Declaration*⟩≡
```
void flip( unsigned int index, bool update = true );
```

The flip method itself simply calls the `getNeighbors()` method of the cube to determine the neighbors. Then, it flips the cell at `index` and at each of the neighbors of that cell.

⟨*FlipFlop Flip Implementation*⟩≡
```
void
⟨FlipFlopNameSpace⟩::flip(
        unsigned int index, bool update
    )
{
    unsigned int nn[ 2 * ⟨CubeNameSpace⟩::DIMENSIONS ];
    unsigned int nc = this->cube->getNeighbors(
         nn, index, this->dims, this->wrap
        );

    ⟨FlipFlop Increment Move Count⟩
    ⟨FlipFlop Flip Single Cell⟩

    for ( unsigned int ii=0; ii < nc; ++ii ) {
        index = nn[ ii ];
        ⟨FlipFlop Flip Single Cell⟩
    }

    ⟨FlipFlop Check Winning Condition⟩
}
```

When flipping a single cell, we must track whether this increased or decreased the number of "on" cells on the board.

⟨*FlipFlop Flip Single Cell*⟩≡
```
if ( ( (*this->cube)[ index ] ^= 1 ) == 0 ) {
    --this->onCount;
} else {
    ++this->onCount;
}
```

Additionally, if we have been asked to update the cell's image on the screen and we've been supplied a view to update, then we will tell the view to redraw the element at this index.

⟨*FlipFlop Flip Single Cell*⟩+≡
```
if ( update && this->view != 0 ) {
    this->view->redraw( index );
}
```

We increment the actual number of moves the user has taken so long as the user has not already won. Either way, if we are updating, and we have a view, we'll play the move noise.

⟨*FlipFlop Increment Move Count*⟩≡
```
if ( ! this->hasWon ) {
    ++this->actualMoves;
}
if ( update && this->view != 0 ) {
    this->view->moveNoise();
}
```

Once all the things have been flipped for this, we have to check the winning condition. However, we only check this during `update` mode. If it's not during `update`, then we stand of chance of congratulating the player while we're still setting up the board. But, if we're in update mode, the user hasn't won, and the number of on pieces on the board is currently zero, that's a victory.

⟨*FlipFlop Check Winning Condition*⟩≡
```
if ( update && ! this->hasWon && this->onCount == 0 ) {
    this->hasWon = true;
    if ( this->view != 0 ) {
        this->view->showWinning(
                this->actualMoves, this->expectedMoves
            );
    }
}
```

10.4 The `FlipFlop` class

In this section, we assemble the `FlipFlop` class from the pieces in the sections above.

The first thing incorporated into the class definition is the declaration of the constructor

⟨*FlipFlop Class Definition*⟩≡
```
public:
    ⟨FlipFlop Constructor Declaration⟩
```

After that, the reset method is declared.

⟨*FlipFlop Class Definition*⟩+≡
```
public:
    ⟨FlipFlop Reset Declaration⟩
```

After that, the flip method is declared.

⟨*FlipFlop Class Definition*⟩+≡
```
public:
    ⟨FlipFlop Flip Declaration⟩
```

The data members of the `FlipFlop` class all have private scope. The data members specify the cube, the number of dimensions, the skill level, the wrapping mode, the number of on elements currently, the variables for tracking the winning state, and the pointer to the view class if one was given.

⟨*FlipFlop Class Definition*⟩+≡
```
private:
    ⟨FlipFlop Cube⟩
    ⟨FlipFlop Dimensions⟩
    ⟨FlipFlop Skill Level⟩
    ⟨FlipFlop Wrap⟩
    ⟨FlipFlop On Count⟩
    ⟨FlipFlop Has Won⟩
    ⟨FlipFlop Move Counters⟩
    ⟨FlipFlop View⟩
```

Once these declarations are all done, we throw all of these together into the class declaration itself.

⟨*FlipFlop Class Declaration*⟩≡
```
class FlipFlop {
    ⟨FlipFlop Class Definition⟩
};
```

10.5 The `flipflop.h` file

In this section, we assemble the header file for the `FlipFlop` class. It is really straightforward since we assembled the class declaration in the previous section. The only thing that we add to the class declaration is that we tuck it into our own name space so that we can keep the global namespace squeaky clean.

⟨*flipflop.h*⟩≡

```
namespace ⟨NameSpace⟩ {
    ⟨FlipFlop Class Declaration⟩
};
```

10.6 The `flipflop.cpp` file

For the actual C++ source code, we include the header file that defines `assert()`, the header file for `random()`, the header file for the `SDL` stuff needed by the `view.h` file, the header file for the cube, the header file for the generic view class, the header file for the view class for this particular game, and the header file generated in the previous section.

⟨*flipflop.cpp*⟩≡

```
#include <assert.h>
#include <stdlib.h>
#include <SDL.h>
#include "cube.h"
#include "soundDev.h"
#include "view.h"
#include "flipflopView.h"
#include "flipflop.h"
```

Then, the source file incorporates the implementation of the constructor.

⟨*flipflop.cpp*⟩+≡

```
⟨FlipFlop Constructor Implementation⟩
```

After that, the source file incorporates the implementation of the reset method.

⟨*flipflop.cpp*⟩+≡

```
⟨FlipFlop Reset Implementation⟩
```

The source file also contains the implementation of the flip method.

⟨*flipflop.cpp*⟩+≡

```
⟨FlipFlop Flip Implementation⟩
```

11 The FlipFlop Game Controller

The namespace inside the FlipFlop controller class is a concatenation of the general namespace and the name of the FlipFlop controller class.

⟨*FlipFlopCNameSpace*⟩≡
```
⟨NameSpace⟩::FlipFlopController
```

The FlipFlop game controller inherits from the generic game controller of §5. It actually controls the initialization and game action of the FlipFlop game. It fields the mouse clicks and converts them from screen coordinates into cell coordinates. And, it fields events from the view sidebar that set the difficulty level and set the wrap mode and set the dimensions and reset the game.

The FlipFlop game controller contains an instance of the FlipFlop game view.

⟨*FlipFlopC View*⟩≡
```
FlipFlopView view;
```

The FlipFlop game controller also contains a pointer to the current instance of the game model.

⟨*FlipFlopC Model*⟩≡
```
FlipFlop* model;
```

11.1 The Constructor and Destructor

The constructor for the FlipFlop controller class takes six arguments. The first is a pointer to the screen, the second is a pointer to the sound device, the third is a pointer to the game cube, the fourth specifies the number of dimensions to employ, the fifth specifies the skill level to use, and the last specifies whether the edges wrap around.

⟨*FlipFlopC Constructor Declaration*⟩≡
```
FlipFlopController(
        SDL_Surface* _screen,
        SoundDev* _sound,
        Cube* _cube,
        unsigned int _dims = 2,
        unsigned int _skillLevel = 0,
        bool _wrap = true
    );
```

The constructor for the FlipFlop controller class simply passes most of its arguments to the `Controller` constructor. Then, it calls its own `reset()` method to allocate a new instance of the `FlipFlop` class.

⟨*FlipFlopC Constructor Implementation*⟩≡

```
⟨FlipFlopCNameSpace⟩::FlipFlopController(
        SDL_Surface* _screen,
        SoundDev* _sound,
        Cube* _cube,
        unsigned int _dims,
        unsigned int _skillLevel,
        bool _wrap
    ) : Controller( _cube, _dims, _skillLevel, _wrap ),
        view( _screen, _sound, _cube, _dims, _skillLevel, _wrap ),
        model( 0 )
{
    this->view.backgroundMusic();
    this->reset();
}
```

The destructor for the FlipFlop controller class deletes the stored model for the FlipFlop game.

⟨*FlipFlopC Destructor Declaration*⟩≡

```
virtual ~FlipFlopController( void );
```

⟨*FlipFlopC Destructor Implementation*⟩≡

```
⟨FlipFlopCNameSpace⟩::~FlipFlopController( void )
{
    this->view.backgroundMusic( true );
    delete this->model;
}
```

11.2 The Reset Method

The `FlipFlopController` class has a method called `reset()`. It uses this method to create a new instance of the FlipFlop game model.

⟨*FlipFlopC Reset Declaration*⟩≡

```
void reset( void );
```

The method first deletes the old model and then creates a new model.

⟨*FlipFlopC Reset Implementation*⟩≡

```
void
⟨FlipFlopCNameSpace⟩::reset( void )
{
    delete this->model;
    this->model = new FlipFlop(
            this->cube,
            this->dims,
            this->skillLevel,
            this->wrap,
            &this->view
        );
}
```

11.3 The Mouse Event Interface

The routine which handles mouse events needs to know whether the event is a mouse press or mouse release. It also needs to know where the event happened. And, it needs to know which mouse button was pressed.

⟨*FlipFlopC Mouse Click Declaration*⟩≡

```
virtual void handleMouseClick(
        bool isMouseUp,
        unsigned int xx,
        unsigned int yy,
        unsigned int buttonNumber
    );
```

This method first gives the mouse click to the view to see if any of the buttons on the sidebar can account for the click. Then, if the view class didn't suck it up, it uses a view method to try to determine which cell of the cube was clicked (if any). If there was a hit, and this is a mouse-down event, then the cell is toggled.

⟨*FlipFlopC Mouse Click Implementation*⟩≡

```
void
⟨FlipFlopCNameSpace⟩::handleMouseClick(
        bool isMouseUp,
        unsigned int xx,
        unsigned int yy,
        unsigned int buttonNumber
    )
{
    unsigned int index;
    bool hit;

    hit = this->view.handleMouseClick(
                this, isMouseUp, xx, yy, buttonNumber
            );

    if ( !hit ) {
        hit = ⟨ViewNameSpace⟩::screenToCell(
                xx, yy, this->dims, &index
            );

        if ( hit && ! isMouseUp ) {
            this->model->flip( index );
        }
    }
}
```

11.4 The Game Setting Interface

The following method is invoked by the `View` class when someone clicks one of the "dimensions" buttons on the sidebar. If the button wasn't already selected, then this triggers a `reset()`.

⟨*FlipFlopC Game Setting Interface*⟩≡

```
virtual void setDimension( unsigned int _dims );
```

⟨*FlipFlopC Game Setting Implementation*⟩≡
```
void
⟨FlipFlopCNameSpace⟩::setDimension(
        unsigned int _dims
    )
{
    if ( _dims != this->dims ) {
        this->dims = _dims;
        this->reset();
    }
}
```

The following method is invoked by the `View` class when someone clicks one of the "skill level" buttons on the sidebar. If the button wasn't already selected, then this triggers a `reset()`.

⟨*FlipFlopC Game Setting Interface*⟩+≡
```
virtual void setSkillLevel( unsigned int _skillLevel );
```

⟨*FlipFlopC Game Setting Implementation*⟩+≡
```
void
⟨FlipFlopCNameSpace⟩::setSkillLevel(
        unsigned int _skillLevel
    )
{
    if ( _skillLevel != this->skillLevel ) {
        this->skillLevel = _skillLevel;
        this->reset();
    }
}
```

The following method is invoked by the `View` class when someone clicks on the "wrap" button on the sidebar. If the button wasn't already selected, then this triggers a `reset()`.

⟨*FlipFlopC Game Setting Interface*⟩+≡
```
virtual void setWrap( bool _wrap );
```

⟨*FlipFlopC Game Setting Implementation*⟩+≡
```
void
⟨FlipFlopCNameSpace⟩::setWrap(
        bool _wrap
    )
{
    if ( _wrap != this->wrap ) {
        this->wrap = _wrap;
        this->reset();
    }
}
```

The following method is invoked by the `View` class when someone clicks on the "new" button on the sidebar. This always triggers a `reset()`.

⟨*FlipFlopC Game Setting Interface*⟩+≡
```
virtual void newGame( void );
```

⟨*FlipFlopC Game Setting Implementation*⟩+≡
```
void
⟨FlipFlopCNameSpace⟩::newGame( void )
{
    this->reset();
}
```

11.5 The `FlipFlopController` class

In this section, we assemble the `FlipFlopController` class from the pieces in the sections above.

We include, in the `FlipFlopController` class, the constructor and the destructor.

⟨*FlipFlopC Class Definition*⟩≡
```
public:
    ⟨FlipFlopC Constructor Declaration⟩
    ⟨FlipFlopC Destructor Declaration⟩
```

The `FlipFlopController` class also declares its reset method and the methods used by the `View` class to change the game state.

⟨*FlipFlopC Class Definition*⟩+≡
```
private:
    ⟨FlipFlopC Reset Declaration⟩
public:
    ⟨FlipFlopC Game Setting Interface⟩
```

We include, in the `FlipFlopController` class, the method used for mouse clicks.

⟨*FlipFlopC Class Definition*⟩+≡
```
public:
    ⟨FlipFlopC Mouse Click Declaration⟩
```

The `FlipFlopController` class also contains the member variables which were defined at the beginning of this section of the document.

⟨*FlipFlopC Class Definition*⟩+≡
```
private:
    ⟨FlipFlopC View⟩
    ⟨FlipFlopC Model⟩
```

Once these declarations are all done, we throw all of these together into the class declaration itself. The `FlipFlopController` inherits directly from the `Controller` class of §5.

⟨*FlipFlopC Class Declaration*⟩≡
```
class FlipFlopController : public Controller {
    ⟨FlipFlopC Class Definition⟩
};
```

11.6 The `flipflopController.h` file

In this section, we assemble the header file for the `FlipFlopController` class. It is really straightforward since we assembled the class declaration in the previous section. The only thing that we add to the class declaration is that we tuck it into our own name space so that we can keep the global namespace squeaky clean.

⟨*flipflopController.h*⟩≡
```
namespace ⟨NameSpace⟩ {
    ⟨FlipFlopC Class Declaration⟩
};
```

11.7 The `flipflopController.cpp` file

In this section, we assemble the FlipFlop controller source file. It requires the header files for the `Cube` class, the `Controller` class, and the `FlipFlopController` classes.

⟨*flipflopController.cpp*⟩≡
```
#include <SDL.h>
#include "cube.h"
#include "soundDev.h"
#include "controller.h"
#include "view.h"
#include "flipflopView.h"
#include "flipflop.h"
#include "flipflopController.h"
```

After the header files, we include the implementations of the constructor and destructor.

⟨*flipflopController.cpp*⟩+≡
```
⟨FlipFlopC Constructor Implementation⟩
⟨FlipFlopC Destructor Implementation⟩
```

After the constructor and destructor, the implementation of the `reset()` method and the game state methods are also included.

⟨*flipflopController.cpp*⟩+≡
```
⟨FlipFlopC Reset Implementation⟩
⟨FlipFlopC Game Setting Implementation⟩
```

Then, we include the implementation of the method used to field mouse clicks.

⟨*flipflopController.cpp*⟩+≡
⟨*FlipFlopC Mouse Click Implementation*⟩

12 The FlipFlop Game View

The namespace inside the FlipFlop view class is a concatenation of the general namespace and the name of the FlipFlop view class.

⟨*FlipFlopVNameSpace*⟩≡
⟨*NameSpace*⟩`::FlipFlopView`

The FlipFlop game view inherits from the generic game view of §6. It displays the current state of the `FlipFlop` game.

The FlipFlop game stores pointers to the images of the tile pieces to use.

⟨*FlipFlop Tiles*⟩≡
```
SDL_Surface* on;
SDL_Surface* off;
```

12.1 The Constructor

The constructor for the FlipFlop view class takes six arguments. The first is a pointer to the screen, the second is a pointer to the sound device, the third is a pointer to the game cube, the fourth specifies the number of dimensions to employ, the fifth is the skill level, and the sixth is the wrapping mode.

⟨*FlipFlopV Constructor Declaration*⟩≡
```
FlipFlopView(
        SDL_Surface* _screen,
        SoundDev* _sound,
        Cube* _cube,
        unsigned int _dims = 2,
        unsigned int _skillLevel = 0,
        bool _wrap = true
    );
```

The constructor for the FlipFlop view class passes all of its arguments to the `View` constructor. Then, it loads the images for on and off cells.

⟨*FlipFlopV Constructor Implementation*⟩≡
```
⟨FlipFlopVNameSpace⟩::FlipFlopView(
        SDL_Surface* _screen,
        SoundDev* _sound,
        Cube* _cube,
        unsigned int _dims,
        unsigned int _skillLevel,
        bool _wrap
    ) : View( _screen, _sound, _cube, _dims, _skillLevel, _wrap )
{
    this->on = ::IMG_Load( "../../data/on.png" );
    this->off = ::IMG_Load( "../../data/off.png" );
}
```

12.2 The Destructor

The destructor for the flip-flop view class simply release the images that it loaded above in the constructor.

⟨*FlipFlopV Destructor Declaration*⟩≡
```
~FlipFlopView( void );
```

⟨*FlipFlopV Destructor Implementation*⟩≡
```
⟨FlipFlopVNameSpace⟩::~FlipFlopView( void )
{
    ::SDL_FreeSurface( this->off );
    ::SDL_FreeSurface( this->on );
}
```

12.3 The Redraw Methods

The FlipFlop view class has a method which allows one to update the entire display area for the game.

⟨*FlipFlopV Redraw Declarations*⟩≡
```
virtual void redraw( void );
```

The redraw function here calls the redraw function on the base class to update the sidebar and the background area of the cube. Then, it runs through each cell in the cube, drawing it. After that, it updates the whole screen.

⟨*FlipFlopV Redraw Implementations*⟩≡
```
void
⟨FlipFlopVNameSpace⟩::redraw( void )
{
    this->View::redraw();

    unsigned int maxIndex
        = ⟨CubeNameSpace⟩::arrayLengths[ this->dims ];

    for ( unsigned int index=0; index < maxIndex; ++index ) {
        this->drawCell( index, false );
    }

    ::SDL_UpdateRect( this->screen, 0, 0, 0, 0 );
}
```

The FlipFlop view class has a method which allows one to update a single cell of the cube by index.

⟨*FlipFlopV Redraw Declarations*⟩+≡
```
virtual void redraw( unsigned int index );
```

This method simply uses the method defined next to draw the single cell in question.

⟨*FlipFlopV Redraw Implementations*⟩+≡

```
void
⟨FlipFlopVNameSpace⟩::redraw( unsigned int index )
{
    this->drawCell( index );
}
```

The FlipFlop view class has a method to draw a single cell of the cube. It uses this method in each of the above methods.

⟨*FlipFlopV Private Draw Declaration*⟩≡

```
void drawCell( unsigned int index, bool update = true );
```

To draw a single cell, this method retrieves the screen coordinates of the cell from the conversion method in the base class. Then, it prepares a rectangle to fill for the cell. Then, depending on the state of the cell in the game cube, it either draws the region off or on.

⟨*FlipFlopV Private Draw Implementation*⟩≡

```
void
⟨FlipFlopVNameSpace⟩::drawCell(
        unsigned int index, bool update
    )
{
    unsigned int xx;
    unsigned int yy;

    View::cellToScreen( index, this->dims, &xx, &yy );

    ⟨FlipFlopV Prepare Single Cell Rect⟩

    if ( (*this->cube)[ index ] == 0 ) {
        ::SDL_BlitSurface( this->off, 0, this->screen, &rect );
    } else {
        ::SDL_BlitSurface( this->on, 0, this->screen, &rect );
    }
    if ( update ) {
        ::SDL_UpdateRect( this->screen, xx, yy, SQUARE, SQUARE );
    }
}
```

The rectangle that will be filled to represent the cell simply starts at the starting coordinates of the rectangle and goes almost the full size of the cell. It doesn't go quite to the edge so that one can clearly see the break between cells.

⟨*FlipFlopV Prepare Single Cell Rect*⟩≡
```
SDL_Rect rect;
rect.x = xx;
rect.y = yy;
rect.w = SQUARE-1;
rect.h = SQUARE-1;
```

12.4 The `FlipFlopView` class

In this section, we assemble the `FlipFlopView` class from the pieces in the sections above.

We include, in the `FlipFlopView` class, the constructor, the destructor and the redraw methods.

⟨*FlipFlopV Class Definition*⟩≡
```
public:
    ⟨FlipFlopV Constructor Declaration⟩
    ⟨FlipFlopV Destructor Declaration⟩
    ⟨FlipFlopV Redraw Declarations⟩
private:
    ⟨FlipFlopV Private Draw Declaration⟩
```

We include the variables that are used in the flipflop view class.

⟨*FlipFlopV Class Definition*⟩+≡
```
private:
    ⟨FlipFlop Tiles⟩
```

Once these declarations are all done, we throw all of these together into the class declaration itself. The `FlipFlopView` inherits directly from the `View` class of §6.

⟨*FlipFlopV Class Declaration*⟩≡
```
class FlipFlopView : public View {
    ⟨FlipFlopV Class Definition⟩
};
```

12.5 The `flipflopView.h` file

In this section, we assemble the header file for the `FlipFlopView` class. It is really straightforward since we assembled the class declaration in the previous section. The only thing that we add to the class declaration is that we tuck it into our own name space so that we can keep the global namespace squeaky clean.

⟨*flipflopView.h*⟩≡
```
namespace ⟨NameSpace⟩ {
    ⟨FlipFlopV Class Declaration⟩
};
```

12.6 The `flipflopView.cpp` file

In this section, we assemble the FlipFlop view source file. It requires the `SDL` headers for dealing with surfaces, the screen, blitting, and loading images. It requires the header files for the `Cube` class, the `SoundDev` class, the `View` class, and the `FlipFlopView` class itself.

⟨*flipflopView.cpp*⟩≡
```
#include <SDL.h>
#include <SDL_image.h>
#include "cube.h"
#include "soundDev.h"
#include "view.h"
#include "flipflopView.h"
```

After the header files, we include the implementations of the constructor, the destructor, and the redraw methods.

⟨*flipflopView.cpp*⟩+≡
```
⟨FlipFlopV Constructor Implementation⟩
⟨FlipFlopV Destructor Implementation⟩
⟨FlipFlopV Redraw Implementations⟩
⟨FlipFlopV Private Draw Implementation⟩
```

Part IV
The Bomb Squad Game

13 Bomb Squad

The namespace inside the bomb squad class is a concatenation of the general namespace and the name of the bomb squad class.

⟨*BombSquadNameSpace*⟩≡
  ⟨*NameSpace*⟩`::BombSquad`

The `BombSquad` class uses several flags to track what has happened in a cell. These flags tell if the cell has been uncovered, if it contains a bomb, and if it contains a flag.

⟨*BombSquad Flags*⟩≡
```
enum {
    UNCOVERED = 0x1000,
    BOMB = 0x2000,
    FLAG = 0x4000
};
```

The `BombSquad` class keeps a pointer to the cube used for the game.

⟨*BombSquad Cube*⟩≡
```
Cube* cube;
```

The `BombSquad` class also tracks the number of dimensions that are being used. It needs this information so that it can properly determine the neighbors of a given point.

⟨*BombSquad Dimensions*⟩≡
```
unsigned int dims;
```

And, the `BombSquad` class tracks the current skill level.

⟨*BombSquad Skill Level*⟩≡
```
unsigned int skillLevel;
```

The `BombSquad` class also keeps track of whether or not it is wrapping around. This is necessary so that it can properly determine neighbors of things near the edge.

⟨*BombSquad Wrap*⟩≡
```
bool wrap;
```

In addition, the game also tracks how many bombs are out there, how many flags have been placed, and how many cells are currently still coverred.

⟨*BombSquad Move Counters*⟩≡
```
unsigned int bombCount;
unsigned int flagCount;
unsigned int coveredCount;
```

The class also tracks whether the game has ended: either because the player won or by a very loud explosion.

⟨*BombSquad Game Over*⟩≡
```
bool gameOver;
```

The `BombSquad` class also keeps track of the view pointer.

⟨*BombSquad View*⟩≡
```
BombSquadView* view;
```

13.1 The Constructor

The constructor for the `BombSquad` class takes five arguments. The first is a pointer to the game cube, the second specifies the number of dimensions to employ, the third specifies the skill level to use, the fourth specifies whether the edges wrap around, and the fifth is an option pointer to the view to update when cells change.

⟨*BombSquad Constructor Declaration*⟩≡
```
BombSquad(
        Cube* _cube,
        unsigned int _dims = 2,
        unsigned int _skillLevel = 0,
        bool _wrap = true,
        BombSquadView* _view = 0
    );
```

The constructor for the `BombSquad` class copies the arguments into its local variables. Then, it calls its own reset method to start a new game. But, first, it verifies that all of the input arguments match its range expectations.

⟨*BombSquad Constructor Implementation*⟩≡

```
⟨BombSquadNameSpace⟩::BombSquad(
        Cube* _cube,
        unsigned int _dims,
        unsigned int _skillLevel,
        bool _wrap,
        BombSquadView* _view
    ) : cube( _cube ),
        dims( _dims ),
        skillLevel( _skillLevel ),
        wrap( _wrap ),
        view( _view )
{
    assert( cube != 0 );
    assert( dims > 1 );
    assert( dims <= ⟨CubeNameSpace⟩::DIMENSIONS );
    assert( ⟨CubeNameSpace⟩::DIMENSIONS <= 4 );
    assert( skillLevel < 3 );
    this->reset();
}
```

13.2 The Reset Method

This method is used to start a new game. It requires no parameters. It assumes that both the number of dimensions and the wrap mode have already been set.

⟨*BombSquad Reset Declaration*⟩≡

```
void reset( void );
```

The cube is cleared. Then, the skill level is used to determine the number of cells to bomb.

⟨*BombSquad Reset Implementation*⟩≡

```
void
⟨BombSquadNameSpace⟩::reset( void )
{
    *this->cube = 0;

    ⟨BombSquad SkillTable⟩
    unsigned int bombs = table[ this->dims ][ this->skillLevel ];
    unsigned int len
        = ⟨CubeNameSpace⟩::arrayLengths[ this->dims ];

    ⟨BombSquad Add Bombs⟩
    ⟨BombSquad Reset Current Statistics⟩

    if ( this->view != 0 ) {
        this->view->reset();
        this->view->redraw();
    }
}
```

The following table is used to determine the number of bombs to place based upon the number of dimensions and the skill level.

⟨*BombSquad SkillTable*⟩≡

```
unsigned int table[ Cube::DIMENSIONS+1 ][ 3 ] = {
    {   0,   0,   0 },
    {   1,   2,   3 },
    {   2,   4,   8 },
    {   4,   8,  16 },
    {  16,  32,  64 },
};
```

We want to be careful not to place a bomb in the same cell twice. So, we make a lookup table of cells we've already bombed. Then, we go through and plant each bomb.

⟨*BombSquad Add Bombs*⟩≡

```
unsigned int* lut = new unsigned int[ bombs ];
unsigned int lutLen = 0;

for ( unsigned int ii=0; ii < bombs; ++ii ) {
    ⟨BombSquad Add Single Bomb⟩
}

delete[] lut;
```

Each time we want to bomb, we pick an index from the available range. Then, we loop through the lookup table incrementing index each time we find a number that is less than or equal to index. This makes index be the `index`-th number which has not yet been chosen. Then, we add the bomb and add index into the lookup table.

⟨*BombSquad Add Single Bomb*⟩≡

```
unsigned int index = random() % len--;
for ( unsigned int jj=0; jj < lutLen; ++jj ) {
    if ( index >= lut[ jj ] ) {
        ++index;
    }
}

(*this->cube)[ index ] |= BOMB;
⟨BombSquad Increment Counts⟩

⟨BombSquad Add Index Into LUT⟩
```

Here, we put it at the end of the lookup table. Then, we keep trying to swap it with the element before it in the list until we come to an element before it that is less than it. We need to keep the lookup table sorted for the previous incrementing loop to work right. This is just a simple bit of a bubble sort where we know that the only element potentially out of place is the last one.

⟨*BombSquad Add Index Into LUT*⟩≡

```
unsigned int spot = lutLen;
while ( spot > 0 && lut[ spot-1 ] > index ) {
    lut[ spot ] = lut[ spot-1 ];
    --spot;
}
lut[ spot ] = index;
++lutLen;
```

Once a bomb has been placed, we increment the counts on each of its neighbors.

⟨*BombSquad Increment Counts*⟩≡

```
unsigned int nn[ 2 * ⟨CubeNameSpace⟩::DIMENSIONS ];
unsigned int nc;

nc = ⟨CubeNameSpace⟩::getNeighbors(
        nn, index, this->dims, this->wrap
    );
for ( unsigned int jj=0; jj < nc; ++jj ) {
    ++(*this->cube)[ nn[ jj ] ];
}
```

Resetting the statistics for the current game is easy. The number of bombs placed is the number that are there. The number of flags placed is zero. And, the number of cells which are covered is the number of cells in this cube.

⟨*BombSquad Reset Current Statistics*⟩≡

```
this->bombCount = bombs;
this->flagCount = 0;
this->coveredCount
    = ⟨CubeNameSpace⟩::arrayLengths[ this->dims ];
this->gameOver = false;
```

13.3 The Uncover Method

This method is used by the controller to reveal a cell. If the cell was entirely empty, then all of its neighbors are also uncovered.

⟨*BombSquad Uncover Declaration*⟩≡

```
void uncover( unsigned int index, bool click = true );
```

If the current cell has not yet been uncovered, it is marked as uncovered. If the cell has no neighboring bombs, then all of its neighbors are uncovered.

⟨*BombSquad Uncover Implementation*⟩≡

```
void
⟨BombSquadNameSpace⟩::uncover(
        unsigned int index, bool click
    )
{
    unsigned int cell = (*this->cube)[ index ];
    if ( ( cell & ( UNCOVERED | FLAG ) ) == 0 ) {
        ⟨BombSquad Register Move⟩
        ⟨BombSquad Uncover Single Cell⟩

        if ( ( cell & BOMB ) != 0 ) {
            ⟨BombSquad Uncover Bomb⟩
        } else if ( ( cell & 0x0FFF ) == 0 ) {
            ⟨BombSquad Uncover Neighbors⟩
        }

        if ( click ) {
            this->checkWinningCondition();
        }
    }
}
```

We update the count of cells which are covered. And, if we have a view and this is for the user's click, then we make a move noise.

⟨*BombSquad Register Move*⟩≡
```
--this->coveredCount;
if ( click && this->view != 0 ) {
    this->view->moveNoise();
}
```

First, we'll mark this particular cell as uncovered.

⟨*BombSquad Uncover Single Cell*⟩≡
```
(*this->cube)[ index ] |= UNCOVERED;
```

Additionally, if we have been supplied a view to update, then we will tell the view to redraw the element at this index.

⟨*BombSquad Uncover Single Cell*⟩+≡
```
if ( this->view != 0 ) {
    this->view->redraw( index );
}
```

If the person uncovered a bomb, then they have lost. If the game isn't already over, then we display the losing message.

⟨*BombSquad Uncover Bomb*⟩≡
```
if ( ! this->gameOver ) {
    this->gameOver = true;
    if ( this->view != 0 ) {
        this->view->showLosing();
    }
}
```

To uncover the neighbors of a cell, we simply retrieve the list of neighbors and uncover each one.

⟨*BombSquad Uncover Neighbors*⟩≡
```
unsigned int nn[ 2 * ⟨CubeNameSpace⟩::DIMENSIONS ];
unsigned int nc = this->cube->getNeighbors(
        nn, index, this->dims, this->wrap
    );

for ( unsigned int ii=0; ii < nc; ++ii ) {
    this->uncover( nn[ ii ], false );
}
```

13.4 The Toggle Flag Method

This method is used by the controller to toggle the flag setting on a cell.

⟨*BombSquad Toggle Flag Declaration*⟩≡
```
void toggleFlag( unsigned int index );
```

This method first checks to make sure that the cell isn't already uncovered. If it isn't, then this method toggles the `FLAG` flag on this cell of the cube. Then, it updates the counter appropriately. Then, it redraws the current cell and checks for a winning condition.

⟨*BombSquad Toggle Flag Implementation*⟩≡

```
void
⟨BombSquadNameSpace⟩::toggleFlag(
        unsigned int index
    )
{
    if ( ( (*this->cube)[ index ] & UNCOVERED ) == 0 ) {
        (*this->cube)[ index ] ^= FLAG;

        ⟨BombSquad ToggleFlag Update Flag Count⟩

        if ( this->view != 0 ) {
            this->view->moveNoise();
            this->view->redraw( index );
        }

        this->checkWinningCondition();
    }
}
```

To update the current count, we have to see if the flag was left in the set or unset position. If the flag was unset, we decrement. If it was set, we increment.

⟨*BombSquad ToggleFlag Update Flag Count*⟩≡

```
if ( ( (*this->cube)[ index ] & FLAG ) == 0 ) {
    --this->flagCount;
} else {
    ++this->flagCount;
}
```

13.5 Checking for a Win

After things have been uncovered or a flag has been planted, we have to check the winning condition. If the game isn't already over, then we have to check to make sure that flag count is the same as the bomb count and the number of cells still covered is the same as the bomb count.

⟨*BombSquad Check Winning Condition Declaration*⟩≡

```
void checkWinningCondition( void );
```

⟨*BombSquad Check Winning Condition Implementation*⟩≡

```
void
⟨BombSquadNameSpace⟩::checkWinningCondition( void ) {
    if ( ! this->gameOver
    && this->flagCount == this->bombCount
    && this->coveredCount == this->bombCount ) {
        this->gameOver = true;
        if ( this->view != 0 ) {
            this->view->showWinning(
                    this->flagCount, this->bombCount
                );
        }
    }
};
```

13.6 The `BombSquad` class

In this section, we assemble the `BombSquad` class from the pieces in the sections above.

The first thing incorporated into the class definition is the declaration of its internal flags.

⟨*BombSquad Class Definition*⟩≡

```
public:
    ⟨BombSquad Flags⟩
```

The next thing incorporated into the class definition is the declaration of the constructor

⟨*BombSquad Class Definition*⟩+≡

```
public:
    ⟨BombSquad Constructor Declaration⟩
```

After that, the reset method is declared.

⟨*BombSquad Class Definition*⟩+≡

```
public:
    ⟨BombSquad Reset Declaration⟩
```

After that, the methods are declared to uncover cells and to toggle the flag.

⟨*BombSquad Class Definition*⟩+≡

```
public:
    ⟨BombSquad Uncover Declaration⟩
    ⟨BombSquad Toggle Flag Declaration⟩
```

Next, the class includes the inline method used to check for a victory.

⟨*BombSquad Class Definition*⟩+≡

```
private:
    ⟨BombSquad Check Winning Condition Declaration⟩
```

The data members of the `BombSquad` class all have private scope. The data members specify the cube, the number of dimensions, the skill level, the wrapping mode, the move counters, the variables for tracking the winning state, and the pointer to the view class if one was given.

⟨*BombSquad Class Definition*⟩+≡

```
private:
    ⟨BombSquad Cube⟩
    ⟨BombSquad Dimensions⟩
    ⟨BombSquad Skill Level⟩
    ⟨BombSquad Wrap⟩
    ⟨BombSquad Move Counters⟩
    ⟨BombSquad Game Over⟩
    ⟨BombSquad View⟩
```

Once these declarations are all done, we throw all of these together into the class declaration itself.

⟨*BombSquad Class Declaration*⟩≡

```
class BombSquad {
    ⟨BombSquad Class Definition⟩
};
```

13.7 The `bomb.h` file

In this section, we assemble the header file for the `BombSquad` class. It is really straightforward since we assembled the class declaration in the previous section. The only thing that we add to the class declaration is that we tuck it into our own name space so that we can keep the global namespace squeaky clean.

⟨*bomb.h*⟩≡

```
namespace ⟨NameSpace⟩ {
    ⟨BombSquad Class Declaration⟩
};
```

13.8 The `bomb.cpp` file

For the actual C++ source code, we include the header file that defines `assert()`, the header file for `random()`, the header file for the `SDL` stuff needed by the `view.h` file, the header file for the cube, the header file for the font, the header file for the generic view class, the header file for the view class for this particular game, and the header file generated in the previous section.

⟨*bomb.cpp*⟩≡

```
#include <assert.h>
#include <stdlib.h>
#include <SDL.h>
#include "cube.h"
#include "font.h"
#include "soundDev.h"
#include "view.h"
#include "bombView.h"
#include "bomb.h"
```

Then, the source file incorporates the implementation of the constructor.

⟨*bomb.cpp*⟩+≡

⟨*BombSquad Constructor Implementation*⟩

After that, the source file incorporates the implementation of the reset method.

⟨*bomb.cpp*⟩+≡

⟨*BombSquad Reset Implementation*⟩

The source file also contains the implementation of the uncover-a-cell method and the toggle-the-flag-state method

⟨*bomb.cpp*⟩+≡

⟨*BombSquad Uncover Implementation*⟩
⟨*BombSquad Toggle Flag Implementation*⟩

The source file also contains the implementation for the method which checks for a win.

⟨*bomb.cpp*⟩+≡

⟨*BombSquad Check Winning Condition Implementation*⟩

14 The BombSquad Game Controller

The namespace inside the BombSquad controller class is a concatenation of the general namespace and the name of the BombSquad controller class.

⟨*BombSquadCNameSpace*⟩≡
⟨*NameSpace*⟩`::BombSquadController`

The BombSquad game controller inherits from the generic game controller of §5. It actually controls the initialization and game action of the BombSquad game. It fields the mouse clicks and converts them from screen coordinates into cell coordinates. And, it fields events from the view sidebar that set the difficulty level and set the wrap mode and set the dimensions and reset the game.

The BombSquad game controller contains an instance of the BombSquad game view.

⟨*BombSquadC View*⟩≡
```
BombSquadView view;
```

The BombSquad game controller also contains a pointer to the current instance of the game model.

⟨*BombSquadC Model*⟩≡
```
BombSquad* model;
```

14.1 The Constructor and Destructor

The constructor for the BombSquad controller class takes six arguments. The first is a pointer to the screen, the second is a pointer to the sound device, the third is a pointer to the game cube, the fourth specifies the number of dimensions to employ, the fifth specifies the skill level to use, and the last specifies whether the edges wrap around.

⟨*BombSquadC Constructor Declaration*⟩≡
```
BombSquadController(
        SDL_Surface* _screen,
        SoundDev* _sound,
        Cube* _cube,
        unsigned int _dims = 2,
        unsigned int _skillLevel = 0,
        bool _wrap = true
    );
```

The constructor for the BombSquad controller class simply passes most of its arguments to the `Controller` constructor. Then, it calls its own `reset()` method to allocate a new instance of the `BombSquad` class.

⟨*BombSquadC Constructor Implementation*⟩≡

```
⟨BombSquadCNameSpace⟩::BombSquadController(
        SDL_Surface* _screen,
        SoundDev* _sound,
        Cube* _cube,
        unsigned int _dims,
        unsigned int _skillLevel,
        bool _wrap
    ) : Controller( _cube, _dims, _skillLevel, _wrap ),
        view( _screen, _sound, _cube, _dims, _skillLevel, _wrap ),
        model( 0 )
{
    this->view.backgroundMusic();
    this->reset();
}
```

The destructor for the BombSquad controller class deletes the stored model for the BombSquad game.

⟨*BombSquadC Destructor Declaration*⟩≡

```
virtual ~BombSquadController( void );
```

⟨*BombSquadC Destructor Implementation*⟩≡

```
⟨BombSquadCNameSpace⟩::~BombSquadController( void )
{
    this->view.backgroundMusic( true );
    delete this->model;
}
```

14.2 The Reset Method

The `BombSquadController` class has a method called `reset()`. It uses this method to create a new instance of the BombSquad game model.

⟨*BombSquadC Reset Declaration*⟩≡

```
void reset( void );
```

The method first deletes the old model and then creates a new model.

⟨*BombSquadC Reset Implementation*⟩≡

```
void
⟨BombSquadCNameSpace⟩::reset( void )
{
    delete this->model;
    this->model = new BombSquad(
            this->cube,
            this->dims,
            this->skillLevel,
            this->wrap,
            &this->view
        );
}
```

14.3 The Mouse Event Interface

The routine which handles mouse events needs to know whether the event is a mouse press or mouse release. It also needs to know where the event happened. And, it needs to know which mouse button was pressed.

⟨*BombSquadC Mouse Click Declaration*⟩≡

```
virtual void handleMouseClick(
        bool isMouseUp,
        unsigned int xx,
        unsigned int yy,
        unsigned int buttonNumber
    );
```

If the hit wasn't in one of the buttons in the view, this method checks to see if the click was a mouse-up event on a cell. If it was a mouse-up event on a cell, then it either uncovers that cell or toggles the flag for that cell depending on which mouse button was used.

⟨*BombSquadC Mouse Click Implementation*⟩≡

```
void
⟨BombSquadCNameSpace⟩::handleMouseClick(
        bool isMouseUp,
        unsigned int xx,
        unsigned int yy,
        unsigned int buttonNumber
    )
{
    unsigned int index;
    bool hit;

    hit = this->view.handleMouseClick(
                this, isMouseUp, xx, yy, buttonNumber
            );

    if ( !hit ) {
        hit = ⟨ViewNameSpace⟩::screenToCell(
                xx, yy, this->dims, &index
            );

        if ( hit && ! isMouseUp ) {
            if ( buttonNumber == 1 ) {
                this->model->uncover( index );
            } else {
                this->model->toggleFlag( index );
            }
        }
    }
}
```

14.4 The Game Setting Interface

The following method is invoked by the `View` class when someone clicks one of the "dimensions" buttons on the sidebar. If the button wasn't already selected, then this triggers a `reset()`.

⟨*BombSquadC Game Setting Interface*⟩≡

```
virtual void setDimension( unsigned int _dims );
```

⟨*BombSquadC Game Setting Implementation*⟩≡

```
void
⟨BombSquadCNameSpace⟩::setDimension(
        unsigned int _dims
    )
{
    if ( _dims != this->dims ) {
        this->dims = _dims;
        this->reset();
    }
}
```

The following method is invoked by the `View` class when someone clicks one of the "skill level" buttons on the sidebar. If the button wasn't already selected, then this triggers a `reset()`.

⟨*BombSquadC Game Setting Interface*⟩+≡

```
virtual void setSkillLevel( unsigned int _skillLevel );
```

⟨*BombSquadC Game Setting Implementation*⟩+≡

```
void
⟨BombSquadCNameSpace⟩::setSkillLevel(
        unsigned int _skillLevel
    )
{
    if ( _skillLevel != this->skillLevel ) {
        this->skillLevel = _skillLevel;
        this->reset();
    }
}
```

The following method is invoked by the `View` class when someone clicks on the "wrap" button on the sidebar. If the button wasn't already selected, then this triggers a `reset()`.

⟨*BombSquadC Game Setting Interface*⟩+≡

```
virtual void setWrap( bool _wrap );
```

⟨*BombSquadC Game Setting Implementation*⟩+≡

```
void
⟨BombSquadCNameSpace⟩::setWrap(
        bool _wrap
    )
{
    if ( _wrap != this->wrap ) {
        this->wrap = _wrap;
        this->reset();
    }
}
```

The following method is invoked by the `View` class when someone clicks on the "new" button on the sidebar. This always triggers a `reset()`.

⟨*BombSquadC Game Setting Interface*⟩+≡
```
virtual void newGame( void );
```

⟨*BombSquadC Game Setting Implementation*⟩+≡
```
void
⟨BombSquadCNameSpace⟩::newGame( void )
{
    this->view.reset();
    this->reset();
}
```

14.5 The `BombSquadController` class

In this section, we assemble the `BombSquadController` class from the pieces in the sections above.

We include, in the `BombSquadController` class, the constructor and the destructor.

⟨*BombSquadC Class Definition*⟩≡
```
public:
    ⟨BombSquadC Constructor Declaration⟩
    ⟨BombSquadC Destructor Declaration⟩
```

The `BombSquadController` class also declares its reset method and the methods used by the `View` class to change the game state.

⟨*BombSquadC Class Definition*⟩+≡
```
private:
    ⟨BombSquadC Reset Declaration⟩
public:
    ⟨BombSquadC Game Setting Interface⟩
```

We include, in the `BombSquadController` class, the method used for mouse clicks.

⟨*BombSquadC Class Definition*⟩+≡
```
public:
    ⟨BombSquadC Mouse Click Declaration⟩
```

The `BombSquadController` class also contains the member variables which were defined at the beginning of this section of the document.

⟨*BombSquadC Class Definition*⟩+≡
```
private:
    ⟨BombSquadC View⟩
    ⟨BombSquadC Model⟩
```

Once these declarations are all done, we throw all of these together into the class declaration itself. The `BombSquadController` inherits directly from the `Controller` class of §5.

⟨*BombSquadC Class Declaration*⟩≡
```
class BombSquadController : public Controller {
    ⟨BombSquadC Class Definition⟩
};
```

14.6 The `bombController.h` file

In this section, we assemble the header file for the `BombSquadController` class. It is really straightforward since we assembled the class declaration in the previous section. The only thing that we add to the class declaration is that we tuck it into our own name space so that we can keep the global namespace squeaky clean.

⟨*bombController.h*⟩≡
```
namespace ⟨NameSpace⟩ {
    ⟨BombSquadC Class Declaration⟩
};
```

14.7 The `bombController.cpp` file

In this section, we assemble the BombSquad controller source file. It requires the header files for the `Cube` class, the `Font` class, the `SoundDev` class, the `Controller` class, the `View` class, the `BombView` class, the `BombSquad` game model, and the `BombSquadController` classes.

⟨*bombController.cpp*⟩≡
```
#include <SDL.h>
#include "cube.h"
#include "font.h"
#include "soundDev.h"
#include "controller.h"
#include "view.h"
#include "bombView.h"
#include "bomb.h"
#include "bombController.h"
```

After the header files, we include the implementations of the constructor and destructor.

⟨*bombController.cpp*⟩+≡
```
⟨BombSquadC Constructor Implementation⟩
⟨BombSquadC Destructor Implementation⟩
```

After the constructor and destructor, the implementation of the `reset()` method and the game state methods are also included.

⟨*bombController.cpp*⟩+≡
⟨*BombSquadC Reset Implementation*⟩
⟨*BombSquadC Game Setting Implementation*⟩

Then, we include the implementation of the method used to field mouse clicks.

⟨*bombController.cpp*⟩+≡
⟨*BombSquadC Mouse Click Implementation*⟩

15 The BombSquad Game View

The namespace inside the BombSquad view class is a concatenation of the general namespace and the name of the BombSquad view class.

⟨*BombSquadVNameSpace*⟩≡
⟨*NameSpace*⟩`::BombSquadView`

The BombSquad game view inherits from the generic game view of §6. It displays the current state of the `BombSquad` game.

The BombSquad game stores pointers to the images of the tile pieces to use.

⟨*BombSquad Tiles*⟩≡
```
SDL_Surface* covered;
SDL_Surface* uncovered;
SDL_Surface* flagged;
SDL_Surface* bomb;
```

The BombSquad game stores a pointer to the font to use for displaying the numbers.

⟨*BombSquad Font*⟩≡
```
Font* font;
```

The BombSquad game also keeps track of whether or not the game is over. If the game is over, then it display bombs whether they are uncovered or not.

⟨*BombSquadV Game Over*⟩≡
```
bool gameOver;
```

15.1 The Constructor

The constructor for the BombSquad view class takes six arguments. The first is a pointer to the screen, the second is a pointer to the sound device, the third is a pointer to the game cube, the fourth specifies the number of dimensions to employ, the fifth is the skill level, and the sixth is the wrapping mode.

⟨*BombSquadV Constructor Declaration*⟩≡
```
BombSquadView(
        SDL_Surface* _screen,
        SoundDev* _sound,
        Cube* _cube,
        unsigned int _dims = 2,
        unsigned int _skillLevel = 0,
        bool _wrap = true
    );
```

The constructor for the BombSquad view class passes all of its arguments to the `View` constructor. Then, it loads the images used to display the tile in the covered or uncovered state and the overlays for flags and bombs. Then, it loads the font so that it can draw the help text in the sidebar.

⟨*BombSquadV Constructor Implementation*⟩≡

```
⟨BombSquadVNameSpace⟩::BombSquadView(
        SDL_Surface* _screen,
        SoundDev* _sound,
        Cube* _cube,
        unsigned int _dims,
        unsigned int _skillLevel,
        bool _wrap
    ) : View( _screen, _sound, _cube, _dims, _skillLevel, _wrap ),
        gameOver( false )
{
    this->covered = ::IMG_Load( "../../data/covered.png" );
    this->uncovered = ::IMG_Load( "../../data/uncovered.png" );
    this->flagged = ::IMG_Load( "../../data/flagged.png" );
    this->bomb = ::IMG_Load( "../../data/bomb.png" );

    this->font = new Font();
}
```

15.2 The Destructor

The destructor for the BombSquad view class simply release the images loaded above in the constructor.

⟨*BombSquadV Destructor Declaration*⟩≡

```
~BombSquadView( void );
```

⟨*BombSquadV Destructor Implementation*⟩≡

```
⟨BombSquadVNameSpace⟩::~BombSquadView( void )
{
    delete this->font;

    ::SDL_FreeSurface( this->bomb );
    ::SDL_FreeSurface( this->flagged );
    ::SDL_FreeSurface( this->uncovered );
    ::SDL_FreeSurface( this->covered );
}
```

15.3 The Redraw Methods

The BombSquad view class has a method which allows one to update the entire display area for the game.

⟨*BombSquadV Redraw Declarations*⟩≡
```
virtual void redraw( void );
```

The redraw function here calls the redraw function on the base class to update the sidebar and the background area of the cube. Then, it runs through each cell in the cube, drawing it. Next, it draws the quick-tip text in the sidebar. After that, it updates the whole screen.

⟨*BombSquadV Redraw Implementations*⟩≡
```
void
⟨BombSquadVNameSpace⟩::redraw( void )
{
    this->View::redraw();

    unsigned int maxIndex
        = ⟨CubeNameSpace⟩::arrayLengths[ this->dims ];

    for ( unsigned int index=0; index < maxIndex; ++index ) {
        this->drawCell( index, false );
    }

    ⟨BombSquadV Draw Quick Tip Text⟩

    ::SDL_UpdateRect( this->screen, 0, 0, 0, 0 );
}
```

In the sidebar, we're going to scribble some hints for the player on what interactions are available.

⟨*BombSquadV Draw Quick Tip Text*⟩≡
```
this->font->centerMessage(
        this->screen, false,
        700, 434,
        "Click on a tile to reveal it."
    );
this->font->centerMessage(
        this->screen, false,
        700, 474,
        "Right-click or Shift-click"
    );
this->font->centerMessage(
        this->screen, false,
        700, 498,
        "on a tile to flag it."
    );
```

The BombSquad view class has a method which allows one to update a single cell of the cube by index.

⟨*BombSquadV Redraw Declarations*⟩+≡
```
virtual void redraw( unsigned int index );
```

This method simply uses the method defined next to draw the single cell in question.

⟨*BombSquadV Redraw Implementations*⟩+≡
```
void
⟨BombSquadVNameSpace⟩::redraw( unsigned int index )
{
    this->drawCell( index );
}
```

The BombSquad view class has a method to draw a single cell of the cube. It uses this method in each of the above methods.

⟨*BombSquadV Private Draw Declaration*⟩≡
```
void drawCell( unsigned int index, bool update = true );
```

To draw a single cell, this method prepares a rectangle to fill for the cell. Then, depending on the state of the cell in the game cube, it either draws it as uncovered or covered. If the game is over or the cell is uncovered, the contents of the cell (a number or bomb) is shown. If the area is flagged, then the flag is drawn over the area.

⟨*BombSquadV Private Draw Implementation*⟩≡

```
void
⟨BombSquadVNameSpace⟩::drawCell(
        unsigned int index, bool update
    )
{
    ⟨BombSquadV Prepare Single Cell Rect⟩

    unsigned int cell = (*this->cube)[ index ];
    bool showContents = this->gameOver;

    if ( ( cell & ⟨BombSquadNameSpace⟩::UNCOVERED ) != 0 ) {
        showContents = true;
        ⟨BombSquadV Draw Cell Uncovered⟩
    } else {
        ⟨BombSquadV Draw Cell Covered⟩
    }

    if ( showContents ) {
        ⟨BombSquadV Draw Cell Contents⟩
    }

    if ( ( cell & ⟨BombSquadNameSpace⟩::FLAG ) != 0 ) {
        ⟨BombSquadV Draw Cell Flag⟩
    }

    if ( update ) {
        ::SDL_UpdateRect( this->screen, xx, yy, SQUARE, SQUARE );
    }
}
```

The rectangle that will be filled to represent the cell simply starts at the starting coordinates of the cell and goes the full size of the cell.

⟨*BombSquadV Prepare Single Cell Rect*⟩≡
```
unsigned int xx;
unsigned int yy;
View::cellToScreen( index, this->dims, &xx, &yy );

SDL_Rect rect;
rect.x = xx;
rect.y = yy;
rect.w = SQUARE;
rect.h = SQUARE;
```

The `uncovered` bitmap is displayed as the backdrop in cells that have already been uncovered.

⟨*BombSquadV Draw Cell Uncovered*⟩≡
```
::SDL_BlitSurface( this->uncovered, 0, this->screen, &rect );
```

The `covered` bitmap is displayed as the backdrop in cells that have not already been uncovered.

⟨*BombSquadV Draw Cell Covered*⟩≡
```
::SDL_BlitSurface( this->covered, 0, this->screen, &rect );
```

If the cell contains a bomb, then its contents are the bomb. If the cell doesn't contain a bomb, but it does have a non-zero number of neighbors which are bombs, then the number is displayed.

⟨*BombSquadV Draw Cell Contents*⟩≡
```
unsigned int count = cell & 0x0FFF;

if ( ( cell & ⟨BombSquadNameSpace⟩::BOMB ) != 0 ) {
    ⟨BombSquadV Draw Cell Bomb⟩
} else if ( count > 0 ) {
    this->font->centerMessage(
            this->screen, false,
            rect.x + rect.w / 2,
            rect.y + rect.h / 2 + 8,
            "%d", count
        );
}
```

If the cell contains a bomb, we overlay the `bomb` bitmap.

⟨*BombSquadV Draw Cell Bomb*⟩≡
```
::SDL_BlitSurface( this->bomb, 0, this->screen, &rect );
```

If the cell is flagged, we overlay the `flagged` bitmap.

⟨*BombSquadV Draw Cell Flag*⟩≡
```
::SDL_BlitSurface( this->flagged, 0, this->screen, &rect );
```

15.4 The Reset Method

The controller needs a way to signal that there is a new game to be played. It invokes the `reset()` method defined here.

⟨*BombSquadV Reset Declaration*⟩≡
```
inline void reset( void ) {
    this->gameOver = false;
    this->View::reset();
};
```

15.5 The Winning Method

Once the game is over, the board must be shown with all of the bombs on it. To accomplish this, this view class overrides the `showWinning()` method to set its own internal flag.

⟨*BombSquadV Winning Declaration*⟩≡
```
virtual void showWinning(
        unsigned int actualMoves,
        unsigned int expectedMoves
    );
```

The method sets its internal flag, then it redraws the screen with this in mind. Then, it displays the default winning message.

⟨*BombSquadV Winning Implementation*⟩≡
```
void
⟨BombSquadVNameSpace⟩::showWinning(
        unsigned int actualMoves,
        unsigned int expectedMoves
    )
{
    this->gameOver = true;
    this->redraw();
    this->View::showWinning( actualMoves, expectedMoves );
}
```

15.6 The Losing Method

Once the game is over, the board must be shown with all of the bombs on it. To accomplish this, this view class overrides the `showLosing()` method to set its own internal flag.

⟨*BombSquadV Losing Declaration*⟩≡
```
virtual void showLosing( void );
```

The method sets its internal flag, then it redraws the screen with this in mind. Then, it displays the default losing message.

⟨*BombSquadV Losing Implementation*⟩≡

```
void
⟨BombSquadVNameSpace⟩::showLosing( void )
{
    this->gameOver = true;
    this->redraw();
    this->View::showLosing();
}
```

15.7 The `BombSquadView` class

In this section, we assemble the `BombSquadView` class from the pieces in the sections above.

We include, in the `BombSquadView` class, the constructor, the destructor, the redraw methods, and the reset method.

⟨*BombSquadV Class Definition*⟩≡

```
public:
    ⟨BombSquadV Constructor Declaration⟩
    ⟨BombSquadV Destructor Declaration⟩
    ⟨BombSquadV Redraw Declarations⟩
    ⟨BombSquadV Reset Declaration⟩
private:
    ⟨BombSquadV Private Draw Declaration⟩
```

The `BombSquadView` class also overrides some of the `View` methods to catch the end-game.

⟨*BombSquadV Class Definition*⟩+≡

```
public:
    ⟨BombSquadV Winning Declaration⟩
    ⟨BombSquadV Losing Declaration⟩
```

We include the variables that are used in the bomb squad view class.

⟨*BombSquadV Class Definition*⟩+≡

```
private:
    ⟨BombSquad Tiles⟩
    ⟨BombSquad Font⟩
    ⟨BombSquadV Game Over⟩
```

Once these declarations are all done, we throw all of these together into the class declaration itself. The `BombSquadView` inherits directly from the `View` class of §6.

⟨*BombSquadV Class Declaration*⟩≡

```
class BombSquadView : public View {
    ⟨BombSquadV Class Definition⟩
};
```

15.8 The bombView.h file

In this section, we assemble the header file for the BombSquadView class. It is really straightforward since we assembled the class declaration in the previous section. The only thing that we add to the class declaration is that we tuck it into our own name space so that we can keep the global namespace squeaky clean.

⟨*bombView.h*⟩≡
```
namespace ⟨NameSpace⟩ {
    ⟨BombSquadV Class Declaration⟩
};
```

15.9 The bombView.cpp file

In this section, we assemble the BombSquad view source file. It requires the SDL headers for dealing with surfaces, the screen, blitting, and loading images. It requires the header files for the Cube class, the Font class, the SoundDev class, the View class, the BombSquadView class itself, and the BombSquad game model.

⟨*bombView.cpp*⟩≡
```
#include <SDL.h>
#include <SDL_image.h>
#include "cube.h"
#include "font.h"
#include "soundDev.h"
#include "view.h"
#include "bombView.h"
#include "bomb.h"
```

After the header files, we include the implementations of the constructor, the destructor, and the redraw methods.

⟨*bombView.cpp*⟩+≡
```
⟨BombSquadV Constructor Implementation⟩
⟨BombSquadV Destructor Implementation⟩
⟨BombSquadV Redraw Implementations⟩
⟨BombSquadV Private Draw Implementation⟩
```

The source file then includes the implementations of the end-game methods that it overrides from the base class.

⟨*bombView.cpp*⟩+≡
```
⟨BombSquadV Winning Implementation⟩
⟨BombSquadV Losing Implementation⟩
```

Part V
The Maze Runner Game

16 Maze Runner

The namespace inside the maze runner class is a concatenation of the general namespace and the name of the maze runner class.

⟨*MazeNameSpace*⟩≡
  ⟨*NameSpace*⟩`::Maze`

The `Maze` class keeps a pointer to the cube used for the game.

⟨*Maze Cube*⟩≡
```
Cube* cube;
```

The `Maze` class uses many constants to keep flags in the cube. First, the `Maze` class has two flags which it only uses during preparation of the maze. The preparation of the maze is described full in §16.2. For now, it suffices to say that it uses the bottom eight bits to tell which set a particular cell belongs to and the ninth bit to tell whether the bottom eight bits name the set or point toward the set.

⟨*Maze Cube Flags*⟩≡
```
enum {
    SET_MASK      = 0x000000FF,
    SET_REFERENCE = 0x00000100
};
```

The `Maze` class also has flags to tell whether a cell has been visited by the player before, where the player currently is, and where the goal position is.

⟨*Maze Cube Flags*⟩+≡
```
enum {
    BEEN_HERE     = 0x00000200,
    AM_HERE       = 0x00000400,
    FINISH_HERE   = 0x00000800
};
```

Additionally, the `Maze` class uses some internal flags to tell which walls are present on a cell. These flags are specificly set up in pairs so that if you have the wall in the negative direction on a particular axis you can shift it up by a bit to get the wall for the positive direction on that same axis. The pairs are ordered from least-significant axis to most-significant axis.

⟨*Maze Cube Flags*⟩+≡

```
enum {
    LEFT          = 0x00010000,
    RIGHT         = 0x00020000,
    UP            = 0x00040000,
    DOWN          = 0x00080000,
    FORE          = 0x00100000,
    AFT           = 0x00200000,
    ANA           = 0x00400000,
    KATA          = 0x00800000,
    ALL_WALLS     = 0x00FF0000
};
```

The `Maze` class also tracks the number of dimensions that are being used. It needs this information so that it can properly determine the neighbors of a given point.

⟨*Maze Dimensions*⟩≡

```
unsigned int dims;
```

And, the `Maze` class tracks the current skill level.

⟨*Maze Skill Level*⟩≡

```
unsigned int skillLevel;
```

The `Maze` class also keeps track of whether or not it is wrapping around. This is necessary so that it can properly determine neighbors of things near the edge.

⟨*Maze Wrap*⟩≡

```
bool wrap;
```

The `Maze` class also keeps track of the index of the current location and the index of the target location.

⟨*Maze Spots*⟩≡

```
unsigned int curIndex;
unsigned int finishIndex;
```

The `Maze` class also keeps track of the number of steps taken and the number of those which were on a spot that was already stepped upon.

⟨*Maze Move Counters*⟩≡

```
int stepsTaken;
int repeatsTaken;
```

Also, to track the winning condition, there is a flag that tells whether the game has already been won or not. After a person wins, she can play around on the board as much as she likes before hitting the "New" button.

⟨*Maze Has Won*⟩≡
```
bool hasWon;
```

The `Maze` class also keeps track of the view pointer.

⟨*Maze View*⟩≡
```
MazeView* view;
```

16.1 The Constructor

The constructor for the `Maze` class takes five arguments. The first is a pointer to the game cube, the second specifies the number of dimensions to employ, the third specifies the skill level to use, the fourth specifies whether the edges wrap around, and the fifth is an option pointer to the view to update when cells change.

⟨*Maze Constructor Declaration*⟩≡
```
Maze(
        Cube* _cube,
        unsigned int _dims = 2,
        unsigned int _skillLevel = 0,
        bool _wrap = true,
        MazeView* _view = 0
    );
```

The constructor for the `Maze` class copies the arguments into its local variables. Then, it calls its own reset method to start a new game. But, first, it verifies that all of the input arguments match its range expectations.

⟨*Maze Constructor Implementation*⟩≡

```
⟨MazeNameSpace⟩::Maze(
        Cube* _cube,
        unsigned int _dims,
        unsigned int _skillLevel,
        bool _wrap,
        MazeView* _view
    ) : cube( _cube ),
        dims( _dims ),
        skillLevel( _skillLevel ),
        wrap( _wrap ),
        view( _view )
{
    assert( cube != 0 );
    assert( dims >= 1 );
    assert( dims <= 4 );
    assert( skillLevel < 3 );
    this->reset();
}
```

16.2 The Reset Method

This method is used to start a new game. It requires no parameters. It assumes that the number of dimensions, the skill level, and the wrap mode have already been set.

⟨*Maze Reset Declaration*⟩≡

```
void reset( void );
```

The cube is set up to contain all walls. The starting set of each member of the cube is itself. This is because the set of a cell is that cell's index if the `SET_REFERENCE` bit is not set. If the `SET_REFERENCE` bit is set, then the set of the cell is the set of the cell whose index is this cell masked with `SET_MASK`. This is described completely in §16.4. Then, this method prepares a list of walls to nuke. Then, we keep trying to nuke walls until there are no more. After that, we reset the game statistics and redraw.

⟨*Maze Reset Implementation*⟩≡

```
void
⟨MazeNameSpace⟩::reset( void )
{
    *this->cube = ALL_WALLS;

    ⟨Maze Extra Wall Table⟩
    unsigned int extraWallPercent
        = table[ this->dims ][ this->skillLevel ];
    unsigned int len
        = ⟨CubeNameSpace⟩::arrayLengths[ this->dims ];

    unsigned int distinctSets = len;
    ⟨Maze Prepare List of Walls⟩

    while ( wallCount > 0 ) {
        unsigned int nn = random() % wallCount;
        ⟨Maze Try To Nuke A Wall⟩
        walls[ nn ] = walls[ wallCount-1 ];
        --wallCount;
    }

    ⟨Maze Reset Current Statistics⟩

    if ( this->view != 0 ) {
        this->view->reset();
        this->view->redraw();
    }
}
```

The following table is used to determine the percentage of extra walls to break down based upon the number of dimensions and the skill level.

⟨*Maze Extra Wall Table*⟩≡
```
unsigned int table[ Cube::DIMENSIONS+1 ][ 3 ] = {
    {  0,  0,  0 },
    { 20, 10,  0 },
    { 20, 10,  0 },
    { 20, 10,  0 },
    { 20, 10,  0 },
};
```

To establish the list of walls, we will make an array big enough to hold every possible wall.

⟨*Maze Prepare List of Walls*⟩≡
```
struct WallInfo {
    unsigned int from;
    unsigned int to;
};
struct WallInfo* walls = new struct WallInfo[
        len * this->dims
    ];
```

Then, we'll go through each cell and add in the walls that we haven't already added.

⟨*Maze Prepare List of Walls*⟩+≡
```
unsigned int wallCount = 0;

for ( unsigned int ii=0; ii < len; ++ii ) {
    unsigned int nn[ 2 * ⟨CubeNameSpace⟩::DIMENSIONS ];
    unsigned int nc = this->cube->getNeighbors(
            nn, ii, this->dims, this->wrap
        );
    for ( unsigned int jj=0; jj < nc; ++jj ) {
        ⟨Maze Add Wall To List⟩
    }
}
```

When adding a potential wall to a list, we only add it if the neighbor has a smaller index than the current cell. In this way, we ensure that we never add the same wall twice.

⟨*Maze Add Wall To List*⟩≡
```
if ( nn[ jj ] < ii ) {
    assert( wallCount < len * this->dims );
    walls[ wallCount ].from = ii;
    walls[ wallCount ].to = nn[ jj ];
    ++wallCount;
}
```

To determine whether we want to nuke a wall, we first see if the cells on opposite sides of the wall are already in the same set. If they are, then we should not nuke the wall unless we're just into randomly nuking it. To nuke the wall, we mark the walls as gone in the cube and then join the sets of the two walls.

⟨*Maze Try To Nuke A Wall*⟩≡
```
unsigned int setFrom = this->set( walls[ nn ].from );
unsigned int setTo = this->set( walls[ nn ].to );

if ( setFrom != setTo || ( random() % 100 ) < extraWallPercent ) {
    ⟨Maze Destroy Wall⟩
    this->join( setFrom, setTo );
}
```

To actually destroy the wall, we first retrieve the coordinates of the original and final cell. Then, we determine which axis the wall is along. Then, we actually erase the walls from the cube.

⟨*Maze Destroy Wall*⟩≡
```
⟨Maze Get To And From Coordinates⟩
⟨Maze Determine Which Axis⟩
⟨Maze Clear Walls⟩
```

We get the coordinates by invoking the `indexToVector()` method of the `Cube` class.

⟨*Maze Get To And From Coordinates*⟩≡
```
unsigned int vf[ ⟨CubeNameSpace⟩::DIMENSIONS ];
⟨CubeNameSpace⟩::indexToVector( walls[ nn ].from, vf );

unsigned int vt[ ⟨CubeNameSpace⟩::DIMENSIONS ];
⟨CubeNameSpace⟩::indexToVector( walls[ nn ].to, vt );
```

Then, we use the helper method defined in §2.3 to tell which axis and which direction along that axis this wall happens to be.

⟨*Maze Determine Which Axis*⟩≡
```
bool positive;
unsigned int axis;

⟨CubeNameSpace⟩::determineAxis(
    vf, vt, this->wrap, &axis, &positive
);
```

To actually clear the walls, we take the lowest possible wall. We shift it over by twice the axis number. Then, for positive differences, we shift the `from` wall one more time. For negative differences, we shift the `to` wall one more time. These walls are erased from the cube.

⟨*Maze Clear Walls*⟩≡
```
unsigned int fw = LEFT << ( 2 * axis );
unsigned int tw = LEFT << ( 2 * axis );

if ( positive ) {
    fw <<= 1;
} else {
    tw <<= 1;
}

(*this->cube)[ walls[ nn ].from ] &= ~fw;
(*this->cube)[ walls[ nn ].to ]   &= ~tw;
```

For example, if the axis were `1` and the difference were positive, then `fw` and `tw` would both be set to `UP` and then `fw` would be shifted to `DOWN`. This means that the lower wall of the `from` location and the upper wall of the `to` location would be erased. This is what one would expect because the cube is presented on-screen with right and down being the positive directions.

To reset the statistics for the game, we first clear out the number of steps taken, the number of steps taken on cells that have already been stepped upon, the winning status, and the current position.

⟨*Maze Reset Current Statistics*⟩≡
```
this->stepsTaken = 0;
this->repeatsTaken = 0;
this->hasWon = false;
this->curIndex = 0;
(*this->cube)[ this->curIndex ] |= AM_HERE;
```

Then, based upon the wrap mode and the current number of dimensions, we decide where the end-point of the maze should be.

⟨*Maze Reset Current Statistics*⟩+≡

```
unsigned int vec[ ⟨CubeNameSpace⟩::DIMENSIONS ];
for ( unsigned int ii=0; ii < ⟨CubeNameSpace⟩::DIMENSIONS; ++ii ) {
    vec[ ii ] = 0;
}

if ( this->wrap ) {
    for ( unsigned int ii=0; ii < this->dims; ++ii ) {
        vec[ ii ] = ⟨CubeNameSpace⟩::SIDE_LENGTH / 2;
    }
} else {
    for ( unsigned int ii=0; ii < this->dims; ++ii ) {
        vec[ ii ] = ⟨CubeNameSpace⟩::SIDE_LENGTH - 1;
    }
}

⟨CubeNameSpace⟩::vectorToIndex( vec, &this->finishIndex );
(*this->cube)[ this->finishIndex ] |= FINISH_HERE;
```

16.3 The Move Method

This method determines whether the person can go in a direct line from the current position to the position given by the `toIndex` parameter.

⟨*Maze Move Declaration*⟩≡

```
void move( unsigned int toIndex );
```

This method saves the current position. Then, it retrieves the coordinates of the current position and the proposed destination. It uses those coordinates to determine which axis is the proposed axis of motion. Then, if there are no obstructions, it makes the actual move.

⟨*Maze Move Implementation*⟩≡

```
void
⟨MazeNameSpace⟩::move( unsigned int toIndex )
{
    unsigned int fromIndex = this->curIndex;
    ⟨Maze Move Get Coordinates⟩
    ⟨Maze Move Determine Axis⟩
    ⟨Maze Check For Walls Along Path⟩
    ⟨Maze Update Counters And Mark Path⟩
}
```

To get the coordinates of the starting and ending locations, we call the `indexToVector` method on the `Cube` class.

⟨*Maze Move Get Coordinates*⟩≡
```
unsigned int vf[ ⟨CubeNameSpace⟩::DIMENSIONS ];
⟨CubeNameSpace⟩::indexToVector( fromIndex, vf );

unsigned int vt[ ⟨CubeNameSpace⟩::DIMENSIONS ];
⟨CubeNameSpace⟩::indexToVector( toIndex, vt );
```

To check the axis of motion, we determine the difference between the current position and the destination position. If the move is not parallel to an axis, we bail.

⟨*Maze Move Determine Axis*⟩≡
```
bool positive;
unsigned int axis;

if ( ⟨CubeNameSpace⟩::determineAxis(
    vf, vt, this->wrap, &axis, &positive
) == false ) {
    return;
}
```

To check for walls along the path, first we're going to copy the starting coordinates into the current coordinates.

⟨*Maze Check For Walls Along Path*⟩≡
```
unsigned int vc[ ⟨CubeNameSpace⟩::DIMENSIONS ];

for ( unsigned int ii=0; ii < ⟨CubeNameSpace⟩::DIMENSIONS; ++ii ) {
    vc[ ii ] = vf[ ii ];
}
```

Then, we're going to step along the axis in the positive direction looking for walls that are in the way.

⟨*Maze Check For Walls Along Path*⟩+≡
```
bool hit = false;

unsigned int wall = RIGHT << ( 2 * axis );
unsigned int positiveDist = 0;

while ( !hit && vc[ axis ] != vt[ axis ] ) {
    hit = ( (*this->cube)[ vc ] & wall ) != 0;
    ⟨Maze Do Positive Step⟩
    ++positiveDist;
}

if ( hit ) {
    positiveDist = 0;
}

vc[ axis ] = vf[ axis ];
```

We're also going to look in the negative direction. If we didn't hit a wall in either, then we'll move the closer direction.

⟨*Maze Check For Walls Along Path*⟩+≡
```
unsigned int negativeDist = 0;

wall = LEFT << ( 2 * axis );

hit = false;

while ( !hit && vc[ axis ] != vt[ axis ] ) {
    hit = ( (*this->cube)[ vc ] & wall ) != 0;
    ⟨Maze Do Negative Step⟩
    ++negativeDist;
}

if ( hit ) {
    negativeDist = 0;
}

vc[ axis ] = vf[ axis ];
```

To move in the positive direction, we simply add one to the current spot on the axis and take the result modulo the size of the cube.

⟨*Maze Do Positive Step*⟩≡
```
vc[ axis ] = ( vc[ axis ] + 1 ) % ⟨CubeNameSpace⟩::SIDE_LENGTH;
```

To move in the negative direction, we simply subtract one from the current spot on the axis and take the result modulo the size of the cube. We have to be careful though not to end up dealing in negative numbers before the modulo though.

⟨*Maze Do Negative Step*⟩≡
```
vc[ axis ] = (
        vc[ axis ] + ⟨CubeNameSpace⟩::SIDE_LENGTH - 1
    ) % ⟨CubeNameSpace⟩::SIDE_LENGTH;
```

If we hit walls in both directions, then this is not a legal move. If it's not a legal move, we'll bail. Otherwise, we'll move whichever way is closer.

⟨*Maze Check For Walls Along Path*⟩+≡
```
if ( positiveDist == 0 && negativeDist == 0 ) {
    return;
}

if ( positiveDist == 0 ) {
    positive = false;
} else if ( negativeDist == 0 ) {
    positive = true;
} else {
    positive = ( positiveDist <= negativeDist );
}
```

If we make it this far, then the move was legal. So, we're going to step through the path, updating our counters and the cells.

⟨*Maze Update Counters And Mark Path*⟩≡
```
(*this->cube)[ this->curIndex ] &= ~AM_HERE;
this->curIndex = toIndex;
(*this->cube)[ this->curIndex ] |= AM_HERE;

while ( vc[ axis ] != vt[ axis ] ) {
    unsigned int index;
    ⟨CubeNameSpace⟩::vectorToIndex( vc, &index );

    ⟨Maze Update Increment Counters⟩
    (*this->cube)[ index ] |= BEEN_HERE;

    if ( this->view != 0 ) {
        this->view->redraw( index );
    }

    if ( positive ) {
        ⟨Maze Do Positive Step⟩
    } else {
        ⟨Maze Do Negative Step⟩
    }
}

if ( this->view != 0 ) {
    this->view->redraw( this->curIndex );
    this->view->moveNoise();
}

⟨Maze Check Winning⟩
```

To update the counters, we have to note that we've taken a step. And, if this is a cell we've been to before, we have to note that the step was a repeat.

⟨*Maze Update Increment Counters*⟩≡
```
++this->stepsTaken;
if ( ( (*this->cube)[ index ] & BEEN_HERE ) != 0 ) {
    ++this->repeatsTaken;
}
```

To check the winning condition, we have to see if we've made it to the finish-point. If we have, then we display the winning message.

⟨*Maze Check Winning*⟩≡
```
if ( !hasWon ) {
    hasWon = ( this->curIndex == this->finishIndex );

    if ( hasWon && this->view != 0 ) {
        this->view->showWinning(
                this->repeatsTaken,
                this->stepsTaken - this->repeatsTaken
            );
    }
}
```

16.4 The Disjoint Set ADT Methods

The maze creation step uses the abstract data type (ADT) for disjoint sets. The basic idea of that ADT is that we have an array which contains integers. If the `SET_REFERENCE` bit *is not* set for a particular item, then the index of that item is the name of its set. If the `SET_REFERENCE` bit *is* set, then the set of this particular item is the same as the set of the item number obtained by masking this item with `SET_MASK`. The code will make it more clear.

There are two set operations defined. The `set()` operation returns the set name for a given index. The `join()` operation joins two sets.

⟨*Maze Set ADT Declarations*⟩≡
```
unsigned int set( unsigned int index );
void join( unsigned int aa, unsigned int bb );
```

This method determines the name of the set for a given index in the cube. If the `SET_REFERENCE` bit is not set, then the name of the set is the same as the index of the cell. If the `SET_REFERENCE` bit is set, then the set is the same as that of the referenced cell. In an effort to keep the depth of the tree small, we always update this cell's set every time we check it.

⟨*Maze Set ADT Implementations*⟩≡
```
unsigned int
⟨MazeNameSpace⟩::set( unsigned int index )
{
    unsigned int value = (*this->cube)[ index ];

    if ( ( value & SET_REFERENCE ) != 0 ) {
        unsigned int ss = this->set( value & SET_MASK );
        ⟨Maze Set Assign Set Number⟩
        return ss;
    } else {
        return index;
    }
}
```

To assign a set number to a cell, we have to clear out all of the bits covered by the `SET_MASK` and then put the set name back in those lower bits.

⟨*Maze Set Assign Set Number*⟩≡
```
assert( ss <= SET_MASK );
value &= ~SET_MASK;
value |= ss;
(*this->cube)[ index ] = value;
```

To join two sets, we simply find the top node in the set of the second guy and make it reference the top node in the set of the first guy.

⟨*Maze Set ADT Implementations*⟩+≡
```
void
⟨MazeNameSpace⟩::join( unsigned int aa, unsigned int bb )
{
    unsigned int ss = this->set( aa );
    unsigned int index = this->set( bb );

    if ( ss != index ) {
        unsigned int value = (*this->cube)[ index ];
        value |= SET_REFERENCE;
        ⟨Maze Set Assign Set Number⟩
    }
}
```

16.5 The `Maze` class

In this section, we assemble the `Maze` class from the pieces in the sections above.

The first thing in the maze class definition is the set of flags used to track the state of cells in the cube.

⟨*Maze Class Definition*⟩≡
```
public:
    ⟨Maze Cube Flags⟩
```

After that is the declaration of the constructor.

⟨*Maze Class Definition*⟩+≡
```
public:
    ⟨Maze Constructor Declaration⟩
```

After that, the reset method and the move method are declared.

⟨*Maze Class Definition*⟩+≡
```
public:
    ⟨Maze Reset Declaration⟩
    ⟨Maze Move Declaration⟩
```

After that, the set abstract data type interfaces are declared.

⟨*Maze Class Definition*⟩+≡
```
private:
    ⟨Maze Set ADT Declarations⟩
```

The data members of the `Maze` class all have private scope. The data members specify the cube, the number of dimensions, the skill level, the wrapping mode, the indexes of the special spots, the move counters, the variables for tracking the winning state, and the pointer to the view class if one was given.

⟨*Maze Class Definition*⟩+≡
```
private:
    ⟨Maze Cube⟩
    ⟨Maze Dimensions⟩
    ⟨Maze Skill Level⟩
    ⟨Maze Wrap⟩
    ⟨Maze Spots⟩
    ⟨Maze Move Counters⟩
    ⟨Maze Has Won⟩
    ⟨Maze View⟩
```

Once these declarations are all done, we throw all of these together into the class declaration itself.

⟨*Maze Class Declaration*⟩≡
```
class Maze {
    ⟨Maze Class Definition⟩
};
```

16.6 The `maze.h` file

In this section, we assemble the header file for the `Maze` class. It is really straightforward since we assembled the class declaration in the previous section. The only thing that we add to the class declaration is that we tuck it into our own name space so that we can keep the global namespace squeaky clean.

⟨*maze.h*⟩≡
```
namespace ⟨NameSpace⟩ {
    ⟨Maze Class Declaration⟩
};
```

16.7 The `maze.cpp` file

For the actual C++ source code, we include the header file that defines `assert()`, the header file for `random()`, the header file for the `SDL` stuff needed by the `view.h` file, the header file for the cube, the header file for the sound device, the header file for the generic view class, the header file for the view class for this particular game, and the header file generated in the previous section.

⟨*maze.cpp*⟩≡
```
#include <assert.h>
#include <stdlib.h>
#include <SDL.h>
#include "cube.h"
#include "soundDev.h"
#include "view.h"
#include "mazeView.h"
#include "maze.h"
```

Then, the source file incorporates the implementation of the constructor.

⟨*maze.cpp*⟩+≡
```
⟨Maze Constructor Implementation⟩
```

After that, the source file incorporates the implementations of the reset method and the move method.

⟨*maze.cpp*⟩+≡
```
⟨Maze Reset Implementation⟩
⟨Maze Move Implementation⟩
```

The source file also contains the implementation of the disjoint set abstract data type.

⟨*maze.cpp*⟩+≡
```
⟨Maze Set ADT Implementations⟩
```

17 The Maze Game Controller

The namespace inside the Maze controller class is a concatenation of the general namespace and the name of the Maze controller class.

⟨*MazeCNameSpace*⟩≡
⟨*NameSpace*⟩`::MazeController`

The Maze game controller inherits from the generic game controller of §5. It actually controls the initialization and game action of the Maze game. It fields the mouse clicks and converts them from screen coordinates into cell coordinates. And, it fields events from the view sidebar that set the difficulty level and set the wrap mode and set the dimensions and reset the game.

The Maze game controller contains an instance of the Maze game view.

⟨*MazeC View*⟩≡
```
MazeView view;
```

The Maze game controller also contains a pointer to the current instance of the game model.

⟨*MazeC Model*⟩≡
```
Maze* model;
```

17.1 The Constructor and Destructor

The constructor for the Maze controller class takes six arguments. The first is a pointer to the screen, the second is a pointer to the sound device, the third is a pointer to the game cube, the fourth specifies the number of dimensions to employ, the fifth specifies the skill level to use, and the last specifies whether the edges wrap around.

⟨*MazeC Constructor Declaration*⟩≡
```
MazeController(
        SDL_Surface* _screen,
        SoundDev* _sound,
        Cube* _cube,
        unsigned int _dims = 2,
        unsigned int _skillLevel = 0,
        bool _wrap = true
    );
```

The constructor for the Maze controller class simply passes most of its arguments to the `Controller` constructor. Then, it calls its own `reset()` method to allocate a new instance of the `Maze` class.

⟨*MazeC Constructor Implementation*⟩≡

```
⟨MazeCNameSpace⟩::MazeController(
        SDL_Surface* _screen,
        SoundDev* _sound,
        Cube* _cube,
        unsigned int _dims,
        unsigned int _skillLevel,
        bool _wrap
    ) : Controller( _cube, _dims, _skillLevel, _wrap ),
        view( _screen, _sound, _cube, _dims, _skillLevel, _wrap ),
        model( 0 )
{
    this->view.backgroundMusic();
    this->reset();
}
```

The destructor for the Maze controller class deletes the stored model for the Maze game.

⟨*MazeC Destructor Declaration*⟩≡

```
virtual ~MazeController( void );
```

⟨*MazeC Destructor Implementation*⟩≡

```
⟨MazeCNameSpace⟩::~MazeController( void )
{
    this->view.backgroundMusic( true );
    delete this->model;
}
```

17.2 The Reset Method

The `MazeController` class has a method called `reset()`. It uses this method to create a new instance of the Maze game model.

⟨*MazeC Reset Declaration*⟩≡

```
void reset( void );
```

The method first deletes the old model and then creates a new model.

⟨*MazeC Reset Implementation*⟩≡

```
void
⟨MazeCNameSpace⟩::reset( void )
{
    delete this->model;
    this->model = new Maze(
            this->cube,
            this->dims,
            this->skillLevel,
            this->wrap,
            &this->view
        );
}
```

17.3 The Mouse Event Interface

The routine which handles mouse events needs to know whether the event is a mouse press or mouse release. It also needs to know where the event happened. And, it gets to know which mouse button was pressed.

⟨*MazeC Mouse Click Declaration*⟩≡

```
virtual void handleMouseClick(
        bool isMouseUp,
        unsigned int xx,
        unsigned int yy,
        unsigned int buttonNumber
    );
```

If the sidebar did not eat up this event, then this method checks to see if the event hit a cell of the cube. If it did and the event was a mouse-release, then this method calls the `move()` method on the model.

⟨*MazeC Mouse Click Implementation*⟩≡

```
void
⟨MazeCNameSpace⟩::handleMouseClick(
        bool isMouseUp,
        unsigned int xx,
        unsigned int yy,
        unsigned int buttonNumber
    )
{
    unsigned int index;
    bool hit;

    hit = this->view.handleMouseClick(
                this, isMouseUp, xx, yy, buttonNumber
            );

    if ( !hit ) {
        hit = ⟨ViewNameSpace⟩::screenToCell(
                xx, yy, this->dims, &index
            );

        if ( hit && ! isMouseUp ) {
            this->model->move( index );
        }
    }
}
```

17.4 The Game Setting Interface

The following method is invoked by the `View` class when someone clicks one of the "dimensions" buttons on the sidebar. If the button wasn't already selected, then this triggers a `reset()`.

⟨*MazeC Game Setting Interface*⟩≡

```
virtual void setDimension( unsigned int _dims );
```

⟨*MazeC Game Setting Implementation*⟩≡
```
void
⟨MazeCNameSpace⟩::setDimension(
        unsigned int _dims
    )
{
    if ( _dims != this->dims ) {
        this->dims = _dims;
        this->reset();
    }
}
```

The following method is invoked by the `View` class when someone clicks one of the "skill level" buttons on the sidebar. If the button wasn't already selected, then this triggers a `reset()`.

⟨*MazeC Game Setting Interface*⟩+≡
```
virtual void setSkillLevel( unsigned int _skillLevel );
```

⟨*MazeC Game Setting Implementation*⟩+≡
```
void
⟨MazeCNameSpace⟩::setSkillLevel(
        unsigned int _skillLevel
    )
{
    if ( _skillLevel != this->skillLevel ) {
        this->skillLevel = _skillLevel;
        this->reset();
    }
}
```

The following method is invoked by the `View` class when someone clicks on the "wrap" button on the sidebar. If the button wasn't already selected, then this triggers a `reset()`.

⟨*MazeC Game Setting Interface*⟩+≡
```
virtual void setWrap( bool _wrap );
```

⟨*MazeC Game Setting Implementation*⟩+≡
```
void
⟨MazeCNameSpace⟩::setWrap(
        bool _wrap
    )
{
    if ( _wrap != this->wrap ) {
        this->wrap = _wrap;
        this->reset();
    }
}
```

The following method is invoked by the `View` class when someone clicks on the "new" button on the sidebar. This always triggers a `reset()`.

⟨*MazeC Game Setting Interface*⟩+≡
```
virtual void newGame( void );
```

⟨*MazeC Game Setting Implementation*⟩+≡
```
void
⟨MazeCNameSpace⟩::newGame( void )
{
    this->reset();
}
```

17.5 The `MazeController` class

In this section, we assemble the `MazeController` class from the pieces in the sections above.

We include, in the `MazeController` class, the constructor and the destructor.

⟨*MazeC Class Definition*⟩≡
```
public:
    ⟨MazeC Constructor Declaration⟩
    ⟨MazeC Destructor Declaration⟩
```

The `MazeController` class also declares its reset method and the methods used by the `View` class to change the game state.

⟨*MazeC Class Definition*⟩+≡
```
private:
    ⟨MazeC Reset Declaration⟩
public:
    ⟨MazeC Game Setting Interface⟩
```

We include, in the `MazeController` class, the method used for mouse clicks.

⟨*MazeC Class Definition*⟩+≡
```
public:
    ⟨MazeC Mouse Click Declaration⟩
```

The `MazeController` class also contains the member variables which were defined at the beginning of this section of the document.

⟨*MazeC Class Definition*⟩+≡
```
private:
    ⟨MazeC View⟩
    ⟨MazeC Model⟩
```

Once these declarations are all done, we throw all of these together into the class declaration itself. The `MazeController` inherits directly from the `Controller` class of §5.

⟨*MazeC Class Declaration*⟩≡
```
class MazeController : public Controller {
    ⟨MazeC Class Definition⟩
};
```

17.6 The `mazeController.h` file

In this section, we assemble the header file for the `MazeController` class. It is really straightforward since we assembled the class declaration in the previous section. The only thing that we add to the class declaration is that we tuck it into our own name space so that we can keep the global namespace squeaky clean.

⟨*mazeController.h*⟩≡
```
namespace ⟨NameSpace⟩ {
    ⟨MazeC Class Declaration⟩
};
```

17.7 The `mazeController.cpp` file

In this section, we assemble the Maze controller source file. It requires the header files for the `Cube` class, the `SoundDev` class, the `Controller` class, the `View` class, the `MazeView` class, the `Maze` class, and the `MazeController` classes.

⟨*mazeController.cpp*⟩≡
```
#include <SDL.h>
#include "cube.h"
#include "soundDev.h"
#include "controller.h"
#include "view.h"
#include "mazeView.h"
#include "maze.h"
#include "mazeController.h"
```

After the header files, we include the implementations of the constructor and destructor.

⟨*mazeController.cpp*⟩+≡
```
⟨MazeC Constructor Implementation⟩
⟨MazeC Destructor Implementation⟩
```

After the constructor and destructor, the implementation of the `reset()` method and the game state methods are also included.

⟨*mazeController.cpp*⟩+≡
```
⟨MazeC Reset Implementation⟩
⟨MazeC Game Setting Implementation⟩
```

Then, we include the implementation of the method used to field mouse clicks.

⟨*mazeController.cpp*⟩+≡
⟨*MazeC Mouse Click Implementation*⟩

18 The Maze Game View

The namespace inside the Maze view class is a concatenation of the general namespace and the name of the Maze view class.

⟨*MazeVNameSpace*⟩≡
⟨*NameSpace*⟩`::MazeView`

The Maze game view inherits from the generic game view of §6. It displays the current state of the `Maze` game.

The Maze game stores pointers to the images of the tile pieces to use. The base image depends on whether the person has been to this spot before or not. The goal position and current position each have their own images which are overlayed on the base image before the walls are displayed. There are twice as many walls as there are dimensions—a wall in the positive and negative direction for each axis. The wall images are overlayed on the backdrop.

⟨*Maze Tiles*⟩≡
```
SDL_Surface* base[ 2 ];
SDL_Surface* finishHere;
SDL_Surface* amHere;
SDL_Surface* walls[ 2 * ⟨CubeNameSpace⟩::DIMENSIONS ];
```

18.1 The Constructor

The constructor for the Maze view class takes six arguments. The first is a pointer to the screen, the second is a pointer to the sound device, the third is a pointer to the game cube, the fourth specifies the number of dimensions to employ, the fifth is the skill level, and the sixth is the wrapping mode.

⟨*MazeV Constructor Declaration*⟩≡
```
MazeView(
        SDL_Surface* _screen,
        SoundDev* _sound,
        Cube* _cube,
        unsigned int _dims = 2,
        unsigned int _skillLevel = 0,
        bool _wrap = true
    );
```

The constructor for the Maze view class passes all of its arguments to the `View` constructor. Then, it loads the images it uses.

⟨*MazeV Constructor Implementation*⟩≡

```
⟨MazeVNameSpace⟩::MazeView(
        SDL_Surface* _screen,
        SoundDev* _sound,
        Cube* _cube,
        unsigned int _dims,
        unsigned int _skillLevel,
        bool _wrap
    ) : View( _screen, _sound, _cube, _dims, _skillLevel, _wrap )
{
    this->base[ 0 ] = ::IMG_Load( "../../data/unmarked.png" );
    this->base[ 1 ] = ::IMG_Load( "../../data/marked.png" );

    this->finishHere = ::IMG_Load( "../../data/goal.png" );
    this->amHere = ::IMG_Load( "../../data/me.png" );

    ⟨MazeV Load Wall Images⟩
}
```

The images for the different walls are simply numbered. There are two walls per dimension—one in the positive direction and one in the negative direction.

⟨*MazeV Load Wall Images*⟩≡

```
for ( unsigned int ii=0; ii < 2 * ⟨CubeNameSpace⟩::DIMENSIONS; ++ii ) {
    char buf[ 64 ];
    sprintf( buf, "../../data/wall%1x.png", ii );
    this->walls[ ii ] = ::IMG_Load( buf );
}
```

Note: The proper way to iterate through these filenames with C++ is to use `strstream` stuff. But, in order to keep the executable very small, I'm not using the standard-template library at all. You could replace the above code with the following code if you were really concerned about staying away from `sprintf(3)`.

⟨*MazeV Proper Load Wall Images*⟩≡

```
for ( unsigned int ii=0; ii < 2 * ⟨CubeNameSpace⟩::DIMENSIONS; ++ii ) {
    std::ostrstream buf;
    buf << "../../data/wall" << hex << ii << std::ends;
    this->walls[ ii ] = ::IMG_Load( buf.str() );
}
```

18.2 The Destructor

The destructor for the maze view class simply release the images that it loaded above in the constructor.

⟨*MazeV Destructor Declaration*⟩≡
```
~MazeView( void );
```

⟨*MazeV Destructor Implementation*⟩≡
```
⟨MazeVNameSpace⟩::~MazeView( void )
{
    ⟨MazeV Release Wall Images⟩

    ::SDL_FreeSurface( this->amHere );
    ::SDL_FreeSurface( this->finishHere );

    ::SDL_FreeSurface( this->base[ 1 ] );
    ::SDL_FreeSurface( this->base[ 0 ] );
}
```

We release the walls in the opposite order they were allocated in an effort to be nice to the memory allocation system.

⟨*MazeV Release Wall Images*⟩≡
```
for ( unsigned int ii=2*⟨CubeNameSpace⟩::DIMENSIONS; ii > 0 ; --ii ) {
    ::SDL_FreeSurface( this->walls[ ii-1 ] );
}
```

18.3 The Redraw Methods

The Maze view class has a method which allows one to update the entire display area for the game.

⟨*MazeV Redraw Declarations*⟩≡
```
virtual void redraw( void );
```

The redraw function here calls the redraw function on the base class to update the sidebar and the background area of the cube. Then, it runs through each cell in the cube, drawing it. After that, it updates the whole screen.

⟨*MazeV Redraw Implementations*⟩≡

```
void
⟨MazeVNameSpace⟩::redraw( void )
{
    this->View::redraw();

    unsigned int maxIndex
        = ⟨CubeNameSpace⟩::arrayLengths[ this->dims ];

    for ( unsigned int index=0; index < maxIndex; ++index ) {
        this->drawCell( index, false );
    }

    ::SDL_UpdateRect( this->screen, 0, 0, 0, 0 );
}
```

The Maze view class has a method which allows one to update a single cell of the cube by index.

⟨*MazeV Redraw Declarations*⟩+≡

```
virtual void redraw( unsigned int index );
```

This method simply uses the method defined next to draw the single cell in question.

⟨*MazeV Redraw Implementations*⟩+≡

```
void
⟨MazeVNameSpace⟩::redraw( unsigned int index )
{
    this->drawCell( index );
}
```

The Maze view class has a method to draw a single cell of the cube. It uses this method in each of the above methods.

⟨*MazeV Private Draw Declaration*⟩≡

```
void drawCell( unsigned int index, bool update = true );
```

To draw a single cell, this method retrieves the screen coordinates of the cell from the conversion method in the base class. Then, it prepares a rectangle to fill for the cell. Then, it uses the state of the cell in the game cube to determine how to draw this cell. It draws this cell differently depending on whether it was visited before or not. Then, if this is the goal location, it draws the goal marker. Then, on top of that, it draws the walls between directions. And, if this is the location with the person in it, it draws the person.

⟨*MazeV Private Draw Implementation*⟩≡

```
void
⟨MazeVNameSpace⟩::drawCell(
        unsigned int index, bool update
    )
{
    unsigned int xx;
    unsigned int yy;

    View::cellToScreen( index, this->dims, &xx, &yy );

    ⟨MazeV Prepare Single Cell Rect⟩

    unsigned int value = (*this->cube)[ index ];

    if ( ( value & ⟨MazeNameSpace⟩::BEEN_HERE ) != 0 ) {
        ::SDL_BlitSurface( this->base[ 1 ], 0, this->screen, &rect );
    } else {
        ::SDL_BlitSurface( this->base[ 0 ], 0, this->screen, &rect );
    }

    if ( ( value & ⟨MazeNameSpace⟩::FINISH_HERE ) != 0 ) {
        ::SDL_BlitSurface( this->finishHere, 0, this->screen, &rect );
    }

    ⟨MazeV Draw Walls⟩

    if ( ( value & ⟨MazeNameSpace⟩::AM_HERE ) != 0 ) {
        ::SDL_BlitSurface( this->amHere, 0, this->screen, &rect );
    }

    if ( update ) {
        ::SDL_UpdateRect( this->screen, xx, yy, SQUARE, SQUARE );
    }
}
```

The rectangle that will be filled to represent the cell simply starts at the starting coordinates of the rectangle and goes the full size of the cell.

⟨*MazeV Prepare Single Cell Rect*⟩≡
```
SDL_Rect rect;
rect.x = xx;
rect.y = yy;
rect.w = SQUARE;
rect.h = SQUARE;
```

To draw the walls, we loop through every possible wall. We check the wall in the cell to see if it is set. If it is, we draw it.

⟨*MazeV Draw Walls*⟩≡
```
for ( unsigned int ii=0; ii < 2 * ⟨CubeNameSpace⟩::DIMENSIONS; ++ii ) {
    unsigned int wall = ⟨MazeNameSpace⟩::LEFT << ii;
    if ( ( value & wall ) != 0 ) {
        ::SDL_BlitSurface( this->walls[ ii ], 0, this->screen, &rect );
    }
}
```

18.4 The `MazeView` class

In this section, we assemble the `MazeView` class from the pieces in the sections above.

We include, in the `MazeView` class, the constructor, the destructor and the redraw methods.

⟨*MazeV Class Definition*⟩≡
```
public:
    ⟨MazeV Constructor Declaration⟩
    ⟨MazeV Destructor Declaration⟩
    ⟨MazeV Redraw Declarations⟩
private:
    ⟨MazeV Private Draw Declaration⟩
```

We include the variables that are used in the maze view class.

⟨*MazeV Class Definition*⟩+≡
```
private:
    ⟨Maze Tiles⟩
```

Once these declarations are all done, we throw all of these together into the class declaration itself. The `MazeView` inherits directly from the `View` class of §6.

⟨*MazeV Class Declaration*⟩≡
```
class MazeView : public View {
    ⟨MazeV Class Definition⟩
};
```

18.5 The mazeView.h file

In this section, we assemble the header file for the MazeView class. It is really straightforward since we assembled the class declaration in the previous section. The only thing that we add to the class declaration is that we tuck it into our own name space so that we can keep the global namespace squeaky clean.

⟨*mazeView.h*⟩≡

```
namespace ⟨NameSpace⟩ {
    ⟨MazeV Class Declaration⟩
};
```

18.6 The mazeView.cpp file

In this section, we assemble the Maze view source file. It requires the SDL headers for dealing with surfaces, the screen, blitting, and loading images. It requires the header files for the Cube class, the SoundDev class, the View class, and the MazeView class itself.

⟨*mazeView.cpp*⟩≡

```
#include <SDL.h>
#include <SDL_image.h>
#include "cube.h"
#include "soundDev.h"
#include "view.h"
#include "mazeView.h"
```

The MazeView implementation also needs the definition of the Maze class in order to get the flags used in the cells of the cube.

⟨*mazeView.cpp*⟩+≡

```
#include "maze.h"
```

After the header files, we include the implementations of the constructor, the destructor, and the redraw methods.

⟨*mazeView.cpp*⟩+≡

```
⟨MazeV Constructor Implementation⟩
⟨MazeV Destructor Implementation⟩
⟨MazeV Redraw Implementations⟩
⟨MazeV Private Draw Implementation⟩
```

Part VI
The Peg Jumper Game

19 Peg Jumper

The namespace inside the peg jumper class is a concatenation of the general namespace and the name of the peg jumper class.

⟨*PegNameSpace*⟩≡
  ⟨*NameSpace*⟩`::Peg`

The `Peg` class defines some constants it uses to keep track of the board state.

⟨*Peg Constants*⟩≡
```
enum {
    EMPTY = 0,
    HOLE = 1,
    PEG = 2,
    SELECTED = 4
};
```

The `Peg` class keeps a pointer to the cube used for the game.

⟨*Peg Cube*⟩≡
```
Cube* cube;
```

The `Peg` class also tracks the number of dimensions that are being used. It needs this information so that it can properly determine the neighbors of a given point.

⟨*Peg Dimensions*⟩≡
```
unsigned int dims;
```

And, the `Peg` class tracks the current skill level.

⟨*Peg Skill Level*⟩≡
```
unsigned int skillLevel;
```

The `Peg` class also keeps track of whether or not it is wrapping around. This is necessary so that it can properly determine neighbors of things near the edge.

⟨*Peg Wrap*⟩≡
```
bool wrap;
```

The `Peg` class also keeps track of the index of the current location of the selected spot. If no spot is selected, then this will be set higher than the maximum index for this number of dimensions.

⟨*Peg Selected Spot*⟩≡
```
unsigned int selectedSpot;
```

The `Peg` class also keeps track of the number of pegs remaining.

⟨*Peg Move Counter*⟩≡
```
unsigned int pegsRemaining;
unsigned int stepsTaken;
bool firstMove;
```

Also, to track the winning condition, there is a flag that tells whether the game has already been won or not. After a person wins, she has to hit the "New" button before getting another game.

⟨*Peg Has Won*⟩≡
```
bool hasWon;
```

The `Peg` class also keeps track of the view pointer.

⟨*Peg View*⟩≡
```
PegView* view;
```

19.1 The Constructor

The constructor for the `Peg` class takes five arguments. The first is a pointer to the game cube, the second specifies the number of dimensions to employ, the third specifies the skill level to use, the fourth specifies whether the edges wrap around, and the fifth is an option pointer to the view to update when cells change.

⟨*Peg Constructor Declaration*⟩≡
```
Peg(
        Cube* _cube,
        unsigned int _dims = 2,
        unsigned int _skillLevel = 0,
        bool _wrap = true,
        PegView* _view = 0
    );
```

The constructor for the `Peg` class copies the arguments into its local variables. Then, it calls its own reset method to start a new game. But, first, it verifies that all of the input arguments match its range expectations.

⟨*Peg Constructor Implementation*⟩≡

```
⟨PegNameSpace⟩::Peg(
        Cube* _cube,
        unsigned int _dims,
        unsigned int _skillLevel,
        bool _wrap,
        PegView* _view
    ) : cube( _cube ),
        dims( _dims ),
        skillLevel( _skillLevel ),
        wrap( _wrap ),
        view( _view )
{
    assert( cube != 0 );
    assert( dims >= 2 );
    assert( dims <= 4 );
    assert( skillLevel < 3 );
    this->reset();
}
```

19.2 The Reset Method

This method is used to start a new game. It requires no parameters. It assumes that the number of dimensions, the skill level, and the wrap mode have already been set.

⟨*Peg Reset Declaration*⟩≡

```
void reset( void );
```

The cube is set up from a pattern given in a data file. Which data file depends upon the skill level and the number of dimensions. Then, the statistics for the game are reset and the view is refreshed.

⟨*Peg Reset Implementation*⟩≡

```
void
⟨PegNameSpace⟩::reset( void )
{
    ⟨Peg Load Board⟩
    ⟨Peg Reset Current Statistics⟩

    if ( this->view != 0 ) {
        this->view->reset();
        this->view->redraw();
    }
}
```

The board to load is chosen based upon the current number of dimensions and the current skill level.

⟨*Peg Load Board*⟩≡

```
char buf[ 512 ];
::sprintf( buf, "../../data/b%d-%d%c.peg",
        this->dims, this->skillLevel,
        ( this->wrap ) ? 'w' : 'n'
    );
```

The real way to do this in C++ involves an `ostrstream`, but that would add quite a bit of size to my program. As this program is being written for a contest with a size limit, I'm using the `sprintf()` call above instead of the block below with uses the standard template library.

⟨*Peg Load Board with STL*⟩≡

```
std::ostrstream stream;

stream << "../../data/b"
    << this->dims << '-'
    << this->skillLevel
    << ( (this->wrap) ? 'w' : 'n' )
    << std::ends;
```

And then the `fopen()` call below would take the argument `stream.str()` instead of `buf`.

The board is read in a space at a time. The board is specified with dashes indicating empty cells, `o`'s indicating holes, and `x`'s indicating pegs.

⟨*Peg Load Board*⟩+≡

```
this->pegsRemaining = 0;

FILE* fp = ::fopen( buf, "r" );
if ( fp != 0 ) {
    for ( unsigned int ii=0; /**/; /**/ ) {
        char ch;
        if ( ::fscanf( fp, "%c", &ch ) == 1 ) {
            if ( ch == '-' ) {
                (*this->cube)[ ii ] = EMPTY;
                ++ii;
            } else if ( ch == 'o' ) {
                (*this->cube)[ ii ] = HOLE;
                ++ii;
            } else if ( ch == 'x' ) {
                (*this->cube)[ ii ] = PEG;
                ++this->pegsRemaining;
                ++ii;
            }
        } else {
            break;
        }
    }

    ::fclose( fp );
}
```

Then, we reset the game statistics. In this case, it's simply that the game has not yet been won.

⟨*Peg Reset Current Statistics*⟩≡

```
this->hasWon = false;
this->firstMove = true;

unsigned int len = ⟨CubeNameSpace⟩::arrayLengths[ this->dims ];
this->selectedSpot = len;
```

19.3 The Check Selected Method

This method checks to see if the selected spot is within the range of the board.

⟨*Peg Check Selected Declaration*⟩≡

```
bool isSelected( void ) const;
```

⟨*Peg Check Selected Implementation*⟩≡

```
bool
⟨PegNameSpace⟩::isSelected( void ) const
{
    unsigned int len = ⟨CubeNameSpace⟩::arrayLengths[ this->dims ];
    return ( this->selectedSpot < len );
}
```

19.4 The Select Method

This method is used to select the peg with which to jump.

⟨*Peg Select Declaration*⟩≡

```
void select( unsigned int cell );
```

This method checks to see if the cell contains a peg. If it does, then whatever cell had been selected is unselected and the new cell is selected.

⟨*Peg Select Implementation*⟩≡

```
void
⟨PegNameSpace⟩::select( unsigned int cell )
{
    unsigned int len = ⟨CubeNameSpace⟩::arrayLengths[ this->dims ];

    if ( ( (*this->cube)[ cell ] & PEG ) != 0 ) {

        ⟨Peg Select Check First Move⟩

        if ( this->selectedSpot < len ) {
            (*this->cube)[ this->selectedSpot ] &= ~SELECTED;
            if ( this->view != 0 ) {
                this->view->redraw( this->selectedSpot );
            }
        }

        (*this->cube)[ cell ] |= SELECTED;
        if ( this->view != 0 ) {
            this->view->moveNoise();
            this->view->redraw( cell );
        }

        this->selectedSpot = cell;
    }
}
```

If this is the first move of the game, then we zot the first peg selected.

⟨*Peg Select Check First Move*⟩≡

```
if ( this->firstMove == true ) {
    (*this->cube)[ cell ] = HOLE;
    if ( this->view != 0 ) {
        this->view->moveNoise();
        this->view->redraw( cell );
    }
    this->firstMove = false;
    --this->pegsRemaining;
    return;
}
```

19.5 The Jump Method

This method is used to jump over the peg specified by the parameter.

⟨*Peg Jump Declaration*⟩≡

```
void jump( unsigned int cell );
```

This method first checks to see if the cell contains a peg. If it does, then it checks to make sure that the selected spot is a neighbor of this spot. If it is, then it checks to make sure that the destination cell is empty and a legitimate neighbor of the current cell. If it is, then the jump is executed.

⟨*Peg Jump Implementation*⟩≡

```
void
⟨PegNameSpace⟩::jump( unsigned int cell )
{
    unsigned int len = ⟨CubeNameSpace⟩::arrayLengths[ this->dims ];
    unsigned int srcSpot = this->selectedSpot;

    ⟨Peg Jump Clear Selected Cell⟩

    ⟨Peg Jump Check For Null Jump⟩
    ⟨Peg Jump Check For Peg⟩
    ⟨Peg Jump Check Selected Is Neighbor⟩
    ⟨Peg Jump Check Destination Empty⟩
    ⟨Peg Jump Check Destination Is Neighbor⟩
    ⟨Peg Jump Do Jump⟩
    ⟨Peg Jump Check Winning⟩
}
```

To clear the selected spot, we mask out all of the bits except the one marked `SELECTED`. Then, we redraw the spot and then reset it so that the `isSelected` method returns false.

⟨*Peg Jump Clear Selected Cell*⟩≡

```
(*this->cube)[ this->selectedSpot ] &= ~SELECTED;
if ( this->view != 0 ) {
    this->view->redraw( this->selectedSpot );
}
this->selectedSpot = len;
```

If the person tried to jump over the spot that was already selected, then we have to just forget the selected spot.

⟨*Peg Jump Check For Null Jump*⟩≡

```
if ( cell == srcSpot ) {
    return;
}
```

If the clicked cell does not contain a peg, then we'll just bail out. We cannot jump an empty cell.

⟨*Peg Jump Check For Peg*⟩≡

```
if ( ( (*this->cube)[ cell ] & PEG ) == 0 ) {
    return;
}
```

Then, we run through the neighbors of the cell looking for the selected spot. If it isn't there, then we'll bail. This is an illegal jump.

⟨*Peg Jump Check Selected Is Neighbor*⟩≡

```
unsigned int nn[ 2 * ⟨CubeNameSpace⟩::DIMENSIONS ];
unsigned int nc = this->cube->getNeighbors(
        nn, cell, this->dims, this->wrap
    );

bool found = false;

for ( unsigned int ii=0; !found && ii < nc; ++ii ) {
    found = ( nn[ ii ] == srcSpot );
}

if ( !found ) {
    return;
}
```

We retrieve the coordinates of the selected cell and the current cell. We use them to determine the coordinates of the destination cell. The idea here is that the difference between the peg to jump over and the peg to jump with should be the same magnitude and sign as the difference between the destination cell and the peg to jump over.

⟨*Peg Jump Check Destination Empty*⟩≡

```
unsigned int src[ ⟨CubeNameSpace⟩::DIMENSIONS ];
unsigned int cur[ ⟨CubeNameSpace⟩::DIMENSIONS ];

⟨CubeNameSpace⟩::indexToVector( srcSpot, src );
⟨CubeNameSpace⟩::indexToVector( cell, cur );

unsigned int dst[ ⟨CubeNameSpace⟩::DIMENSIONS ];

for ( unsigned int ii=0; ii < ⟨CubeNameSpace⟩::DIMENSIONS; ++ii ) {
    dst[ ii ] = ( cur[ ii ]
            + ( ⟨CubeNameSpace⟩::SIDE_LENGTH
                + cur[ ii ] - src[ ii ]
            )
        ) % ⟨CubeNameSpace⟩::SIDE_LENGTH;
}

unsigned int index;
⟨CubeNameSpace⟩::vectorToIndex( dst, &index );

if ( ( (*this->cube)[ index ] & HOLE ) == 0 ) {
    return;
}
```

We also have to make sure that the destination cell is a legitimate neighbor of the current cell. This is necessary because the calculation doesn't care whether wrapping is turned on or not.

⟨*Peg Jump Check Destination Is Neighbor*⟩≡

```
found = false;

for ( unsigned int ii=0; !found && ii < nc; ++ii ) {
    found = ( nn[ ii ] == index );
}

if ( !found ) {
    return;
}
```

To actually do the jump, we clear out the selected cell and the current cell and put a peg in the destination cell.

⟨*Peg Jump Do Jump*⟩≡
```
(*this->cube)[ srcSpot ] = HOLE;
(*this->cube)[ cell ] = HOLE;
(*this->cube)[ index ] = PEG;

if ( this->view != 0 ) {
    this->view->moveNoise();
    this->view->redraw( srcSpot );
    this->view->redraw( cell );
    this->view->redraw( index );
}

--this->pegsRemaining;
++this->stepsTaken;
```

The game has been won if there is only one peg remaining.

⟨*Peg Jump Check Winning*⟩≡
```
if ( !this->hasWon ) {
    this->hasWon = ( this->pegsRemaining == 1 );

    if ( this->hasWon && this->view != 0 ) {
        ⟨Peg Jump Show Winning⟩
    }
}
```

To display the winning, we simply call the appropriate method on the `View` class.

⟨*Peg Jump Show Winning*⟩≡
```
this->view->showWinning(
        this->stepsTaken,
        this->stepsTaken
    );
```

19.6 The `Peg` class

In this section, we assemble the `Peg` class from the pieces in the sections above.

The first thing declared in the `Peg` class are the constants used to label the contents of the cells.

⟨*Peg Class Definition*⟩≡
```
public:
    ⟨Peg Constants⟩
```

The next thing declared in the `Peg` class is the constructor.

⟨*Peg Class Definition*⟩+≡
```
public:
    ⟨Peg Constructor Declaration⟩
```

After that, the reset method, the is-selected method, the select method, and the jump method are declared.

⟨*Peg Class Definition*⟩+≡
```
public:
    ⟨Peg Reset Declaration⟩
    ⟨Peg Check Selected Declaration⟩
    ⟨Peg Select Declaration⟩
    ⟨Peg Jump Declaration⟩
```

The data members of the `Peg` class all have private scope. The data members specify the cube, the number of dimensions, the skill level, the wrapping mode, the current location of the selected spot, the variables for tracking the winning state, and the pointer to the view class if one was given.

⟨*Peg Class Definition*⟩+≡
```
private:
    ⟨Peg Cube⟩
    ⟨Peg Dimensions⟩
    ⟨Peg Skill Level⟩
    ⟨Peg Wrap⟩
    ⟨Peg Selected Spot⟩
    ⟨Peg Move Counter⟩
    ⟨Peg Has Won⟩
    ⟨Peg View⟩
```

Once these declarations are all done, we throw all of these together into the class declaration itself.

⟨*Peg Class Declaration*⟩≡
```
class Peg {
    ⟨Peg Class Definition⟩
};
```

19.7 The `peg.h` file

In this section, we assemble the header file for the `Peg` class. It is really straightforward since we assembled the class declaration in the previous section. The only thing that we add to the class declaration is that we tuck it into our own name space so that we can keep the global namespace squeaky clean.

⟨*peg.h*⟩≡
```
namespace ⟨NameSpace⟩ {
    ⟨Peg Class Declaration⟩
};
```

19.8 The `peg.cpp` file

For the actual C++ source code, we include the header file that defines `assert()`, the header file for `random()`, the header file for the `SDL` stuff needed by the `view.h` file, the header file for the cube, the header for the sound device, the header file for the generic view class, the header file for the view class for this particular game, the header file for the font class, the header file for the peg view class, and the header file generated in the previous section.

⟨*peg.cpp*⟩≡

```
#include <assert.h>
#include <stdlib.h>
#include <SDL.h>
#include "cube.h"
#include "soundDev.h"
#include "view.h"
#include "font.h"
#include "pegView.h"
#include "peg.h"
```

Then, the source file incorporates the implementation of the constructor.

⟨*peg.cpp*⟩+≡

⟨*Peg Constructor Implementation*⟩

After that, the source file incorporates the implementations of the reset method, the is-selected method, the select method, and the jump method.

⟨*peg.cpp*⟩+≡

⟨*Peg Reset Implementation*⟩
⟨*Peg Check Selected Implementation*⟩
⟨*Peg Select Implementation*⟩
⟨*Peg Jump Implementation*⟩

20 The Peg Jumpers View

The namespace inside the peg view class is a concatenation of the general namespace and the name of the peg view class.

⟨*PegVNameSpace*⟩≡
⟨*NameSpace*⟩`::PegView`

The Peg game view inherits from the generic game view of §6. It displays the current state of the `Peg` game.

The Peg game stores pointers to the images containing the tile pieces to use.

⟨*Peg Tiles*⟩≡
```
SDL_Surface* peg;
SDL_Surface* hole;
SDL_Surface* empty;
SDL_Surface* selected;
```

The Peg game stores a pointer to the font to use for displaying the hint text.

⟨*Peg Font*⟩≡
```
Font* font;
```

20.1 The Constructor

The constructor for the Peg view class takes six arguments. The first is a pointer to the screen, the second is a pointer to the sound device, the third is a pointer to the game cube, the fourth specifies the number of dimensions to employ, the fifth is the skill level, and the sixth is the wrapping mode.

⟨*PegV Constructor Declaration*⟩≡
```
PegView(
        SDL_Surface* _screen,
        SoundDev* _sound,
        Cube* _cube,
        unsigned int _dims = 2,
        unsigned int _skillLevel = 0,
        bool _wrap = true
    );
```

The constructor for the Peg view class passes all of its arguments to the `View` constructor. Then, it loads the images it uses. After that, it loads the font.

⟨*PegV Constructor Implementation*⟩≡

```
⟨PegVNameSpace⟩::PegView(
        SDL_Surface* _screen,
        SoundDev* _sound,
        Cube* _cube,
        unsigned int _dims,
        unsigned int _skillLevel,
        bool _wrap
    ) : View( _screen, _sound, _cube, _dims, _skillLevel, _wrap )
{
    assert( this->screen != 0 );
    assert( this->dims <= 4 );

    this->peg = ::IMG_Load( "../../data/peg.png" );
    this->hole = ::IMG_Load( "../../data/hole.png" );
    this->empty = ::IMG_Load( "../../data/empty.png" );
    this->selected = ::IMG_Load( "../../data/selected.png" );

    this->font = new Font();
}
```

20.2 The Destructor

The destructor for the peg view class simply release the images and font that it loaded above in the constructor.

⟨*PegV Destructor Declaration*⟩≡

```
~PegView( void );
```

⟨*PegV Destructor Implementation*⟩≡

```
⟨PegVNameSpace⟩::~PegView( void )
{
    delete this->font;

    ::SDL_FreeSurface( this->selected );
    ::SDL_FreeSurface( this->empty );
    ::SDL_FreeSurface( this->hole );
    ::SDL_FreeSurface( this->peg );
}
```

20.3 The Redraw Methods

The Peg view class has a method which allows one to update the entire display area for the game.

⟨*PegV Redraw Declarations*⟩≡
```
virtual void redraw( void );
```

The redraw function here calls the redraw function on the base class to update the sidebar and the background area of the cube. Then, it runs through each cell in the cube, drawing it. It draws some quick hints in the sidebar. After that, it updates the whole screen.

⟨*PegV Redraw Implementations*⟩≡
```
void
⟨PegVNameSpace⟩::redraw( void )
{
    this->View::redraw();

    unsigned int maxIndex
        = ⟨CubeNameSpace⟩::arrayLengths[ this->dims ];

    for ( unsigned int index=0; index < maxIndex; ++index ) {
        this->drawCell( index, false );
    }

    ⟨PegV Draw Quick Tip Text⟩

    ::SDL_UpdateRect( this->screen, 0, 0, 0, 0 );
}
```

In the sidebar, we're going to scribble some hints for the player on what interactions are available.

⟨*PegV Draw Quick Tip Text*⟩≡
```
this->font->centerMessage(
        this->screen, false,
        700, 384,
        "Remove a peg to start."
    );
this->font->centerMessage(
        this->screen, false,
        700, 434,
        "Click the peg to jump with."
    );
this->font->centerMessage(
        this->screen, false,
        700, 474,
        "Then, click the peg"
    );
this->font->centerMessage(
        this->screen, false,
        700, 498,
        "to jump over."
    );
```

The Peg view class has a method which allows one to update a single cell of the cube by index.

⟨*PegV Redraw Declarations*⟩+≡
```
virtual void redraw( unsigned int index );
```

This method simply uses the method defined next to draw the single cell in question.

⟨*PegV Redraw Implementations*⟩+≡
```
void
⟨PegVNameSpace⟩::redraw( unsigned int index )
{
    this->drawCell( index );
}
```

The Peg view class has a method to draw a single cell of the cube. It uses this method in each of the above methods.

⟨*PegV Private Draw Declaration*⟩≡
```
void drawCell( unsigned int index, bool update = true );
```

To draw a single cell, this method retrieves the screen coordinates of the cell from the conversion method in the base class. Then, it prepares a rectangle to fill for the cell. Then, it checks the state of the cell in the cube to determine how to draw it.

⟨*PegV Private Draw Implementation*⟩≡

```
void
⟨PegVNameSpace⟩::drawCell(
        unsigned int index, bool update
    )
{
    unsigned int xx;
    unsigned int yy;

    View::cellToScreen( index, this->dims, &xx, &yy );

    ⟨PegV Prepare Single Cell Dst Rect⟩

    unsigned int value = (*this->cube)[ index ];

    if ( ( value & ⟨PegNameSpace⟩::PEG ) != 0 ) {
        ::SDL_BlitSurface( this->peg, 0, this->screen, &dst );
    } else if ( ( value & ⟨PegNameSpace⟩::HOLE ) != 0 ) {
        ::SDL_BlitSurface( this->hole, 0, this->screen, &dst );
    } else {
        ::SDL_BlitSurface( this->empty, 0, this->screen, &dst );
    }

    if ( ( value & ⟨PegNameSpace⟩::SELECTED ) != 0 ) {
        ::SDL_BlitSurface( this->selected, 0, this->screen, &dst );
    }

    if ( update ) {
        ::SDL_UpdateRect( this->screen, xx, yy, SQUARE, SQUARE );
    }
}
```

The rectangle that will be filled to represent the cell simply starts at the starting coordinates of the rectangle and goes the full size of the cell.

⟨*PegV Prepare Single Cell Dst Rect*⟩≡

```
SDL_Rect dst;
dst.x = xx;
dst.y = yy;
dst.w = SQUARE;
dst.h = SQUARE;
```

20.4 The PegView class

In this section, we assemble the PegView class from the pieces in the sections above.

We include, in the PegView class, the constructor, the destructor, and the redraw methods.

⟨*PegV Class Definition*⟩≡
```
public:
    ⟨PegV Constructor Declaration⟩
    ⟨PegV Destructor Declaration⟩
    ⟨PegV Redraw Declarations⟩
private:
    ⟨PegV Private Draw Declaration⟩
```

We include the variables that are used in the tile view class.

⟨*PegV Class Definition*⟩+≡
```
private:
    ⟨Peg Tiles⟩
    ⟨Peg Font⟩
```

Once these declarations are all done, we throw all of these together into the class declaration itself. The PegView inherits directly from the View class of §6.

⟨*PegV Class Declaration*⟩≡
```
class PegView : public View {
    ⟨PegV Class Definition⟩
};
```

20.5 The pegView.h file

In this section, we assemble the header file for the PegView class. It is really straightforward since we assembled the class declaration in the previous section. The only thing that we add to the class declaration is that we tuck it into our own name space so that we can keep the global namespace squeaky clean.

⟨*pegView.h*⟩≡
```
namespace ⟨NameSpace⟩ {
    ⟨PegV Class Declaration⟩
};
```

20.6 The pegView.cpp file

In this section, we assemble the Peg view source file. It requires the SDL headers for dealing with surfaces, the screen, blitting, and loading images. It requires the header files for the Cube class, the Font class, the SoundDev class, the View class, the PegView class itself, and the Peg class.

⟨*pegView.cpp*⟩≡

```
#include <assert.h>
#include <SDL.h>
#include <SDL_image.h>
#include "cube.h"
#include "font.h"
#include "soundDev.h"
#include "view.h"
#include "pegView.h"
#include "peg.h"
```

After the header files, we include the implementations of the constructor, the destructor, the redraw methods, and the goal state methods.

⟨*pegView.cpp*⟩+≡

⟨*PegV Constructor Implementation*⟩
⟨*PegV Destructor Implementation*⟩
⟨*PegV Redraw Implementations*⟩
⟨*PegV Private Draw Implementation*⟩

21 The Peg Jumper Game Controller

The namespace inside the peg controller class is a concatenation of the general namespace and the name of the Peg controller class.

⟨*PegCNameSpace*⟩≡
⟨*NameSpace*⟩`::PegController`

The Peg game controller inherits from the generic game controller of §5. It actually controls the initialization and game action of the Peg game. It fields the mouse clicks and converts them from screen coordinates into cell coordinates. And, it fields events from the view sidebar that set the difficulty level and set the wrap mode and set the dimensions and reset the game.

The Peg game controller contains an instance of the Peg game view.

⟨*PegC View*⟩≡
```
PegView view;
```

The Peg game controller also contains a pointer to the current instance of the game model.

⟨*PegC Model*⟩≡
```
Peg* model;
```

21.1 The Constructor and Destructor

The constructor for the Peg controller class takes six arguments. The first is a pointer to the screen, the second is a pointer to the sound device, the third is a pointer to the game cube, the fourth specifies the number of dimensions to employ, the fifth specifies the skill level to use, and the last specifies whether the edges wrap around.

⟨*PegC Constructor Declaration*⟩≡
```
PegController(
        SDL_Surface* _screen,
        SoundDev* _sound,
        Cube* _cube,
        unsigned int _dims = 2,
        unsigned int _skillLevel = 0,
        bool _wrap = true
    );
```

The constructor for the Peg controller class simply passes most of its arguments to the `Controller` constructor. Then, it calls its own `reset()` method to allocate a new instance of the `Peg` class.

⟨*PegC Constructor Implementation*⟩≡

```
⟨PegCNameSpace⟩::PegController(
        SDL_Surface* _screen,
        SoundDev* _sound,
        Cube* _cube,
        unsigned int _dims,
        unsigned int _skillLevel,
        bool _wrap
    ) : Controller( _cube, _dims, _skillLevel, _wrap ),
        view( _screen, _sound, _cube, _dims, _skillLevel, _wrap ),
        model( 0 )
{
    this->view.backgroundMusic();
    this->reset();
}
```

The destructor for the Peg controller class deletes the stored model for the Peg game.

⟨*PegC Destructor Declaration*⟩≡

```
virtual ~PegController( void );
```

⟨*PegC Destructor Implementation*⟩≡

```
⟨PegCNameSpace⟩::~PegController( void )
{
    this->view.backgroundMusic( true );
    delete this->model;
}
```

21.2 The Reset Method

The `PegController` class has a method called `reset()`. It uses this method to create a new instance of the Peg game model.

⟨*PegC Reset Declaration*⟩≡

```
void reset( void );
```

The method first deletes the old model and then creates a new model.

⟨*PegC Reset Implementation*⟩≡
```
void
⟨PegCNameSpace⟩::reset( void )
{
    delete this->model;
    this->model = new Peg(
            this->cube,
            this->dims,
            this->skillLevel,
            this->wrap,
            &this->view
        );
}
```

21.3 The Mouse Event Interface

The routine which handles mouse events needs to know whether the event is a mouse press or mouse release. It also needs to know where the event happened. And, it gets to know which mouse button was pressed.

⟨*PegC Mouse Click Declaration*⟩≡
```
virtual void handleMouseClick(
        bool isMouseUp,
        unsigned int xx,
        unsigned int yy,
        unsigned int buttonNumber
    );
```

This method first gives the base class a chance to absorb the event with the sidebar buttons. If it does not, then this method looks to see if any cell of the cube was hit if the event is a mouse release. If the user right-clicked or if there is no cell currently selected, this method selects the cell. If there is a cell selected and this was a left-click, this method tries to jump the given cell.

⟨*PegC Mouse Click Implementation*⟩≡

```
void
⟨PegCNameSpace⟩::handleMouseClick(
        bool isMouseUp,
        unsigned int xx,
        unsigned int yy,
        unsigned int buttonNumber
    )
{
    unsigned int index;
    bool hit;

    hit = this->view.handleMouseClick(
                this, isMouseUp, xx, yy, buttonNumber
            );

    if ( !hit && !isMouseUp ) {
        hit = ⟨ViewNameSpace⟩::screenToCell(
                xx, yy, this->dims, &index
            );

        if ( hit ) {
            if ( buttonNumber > 1 || !this->model->isSelected() ) {
                this->model->select( index );
            } else {
                this->model->jump( index );
            }
        }
    }
}
```

21.4 The Game Setting Interface

The following method is invoked by the `View` class when someone clicks one of the "dimensions" buttons on the sidebar. If the button wasn't already selected, then this triggers a `reset()`.

⟨*PegC Game Setting Interface*⟩≡

```
virtual void setDimension( unsigned int _dims );
```

⟨*PegC Game Setting Implementation*⟩≡
```
void
⟨PegCNameSpace⟩::setDimension(
        unsigned int _dims
    )
{
    if ( _dims != this->dims ) {
        this->dims = _dims;
        this->reset();
    }
}
```

The following method is invoked by the `View` class when someone clicks one of the "skill level" buttons on the sidebar. If the button wasn't already selected, then this triggers a `reset()`.

⟨*PegC Game Setting Interface*⟩+≡
```
virtual void setSkillLevel( unsigned int _skillLevel );
```

⟨*PegC Game Setting Implementation*⟩+≡
```
void
⟨PegCNameSpace⟩::setSkillLevel(
        unsigned int _skillLevel
    )
{
    if ( _skillLevel != this->skillLevel ) {
        this->skillLevel = _skillLevel;
        this->reset();
    }
}
```

The following method is invoked by the `View` class when someone clicks on the "wrap" button on the sidebar. If the button wasn't already selected, then this triggers a `reset()`.

⟨*PegC Game Setting Interface*⟩+≡
```
virtual void setWrap( bool _wrap );
```

⟨*PegC Game Setting Implementation*⟩+≡
```
void
⟨PegCNameSpace⟩::setWrap(
        bool _wrap
    )
{
    if ( _wrap != this->wrap ) {
        this->wrap = _wrap;
        this->reset();
    }
}
```

The following method is invoked by the `View` class when someone clicks on the "new" button on the sidebar. This always triggers a `reset()`.

⟨*PegC Game Setting Interface*⟩+≡
```
virtual void newGame( void );
```

⟨*PegC Game Setting Implementation*⟩+≡
```
void
⟨PegCNameSpace⟩::newGame( void )
{
    this->reset();
}
```

21.5 The `PegController` class

In this section, we assemble the `PegController` class from the pieces in the sections above.

We include, in the `PegController` class, the constructor and the destructor.

⟨*PegC Class Definition*⟩≡
```
public:
    ⟨PegC Constructor Declaration⟩
    ⟨PegC Destructor Declaration⟩
```

The `PegController` class also declares its reset method and the methods used by the `View` class to change the game state.

⟨*PegC Class Definition*⟩+≡
```
private:
    ⟨PegC Reset Declaration⟩
public:
    ⟨PegC Game Setting Interface⟩
```

We include, in the `PegController` class, the method used for mouse clicks.

⟨*PegC Class Definition*⟩+≡
```
public:
    ⟨PegC Mouse Click Declaration⟩
```

The `PegController` class also contains the member variables which were defined at the beginning of this section of the document.

⟨*PegC Class Definition*⟩+≡
```
private:
    ⟨PegC View⟩
    ⟨PegC Model⟩
```

Once these declarations are all done, we throw all of these together into the class declaration itself. The `PegController` inherits directly from the `Controller` class of §5.

⟨*PegC Class Declaration*⟩≡
```
class PegController : public Controller {
    ⟨PegC Class Definition⟩
};
```

21.6 The `pegController.h` file

In this section, we assemble the header file for the `PegController` class. It is really straightforward since we assembled the class declaration in the previous section. The only thing that we add to the class declaration is that we tuck it into our own name space so that we can keep the global namespace squeaky clean.

⟨*pegController.h*⟩≡
```
namespace ⟨NameSpace⟩ {
    ⟨PegC Class Declaration⟩
};
```

21.7 The `pegController.cpp` file

In this section, we assemble the Peg controller source file. It requires the header files for the `Cube` class the `SoundDev` class, the `Controller` class, the `Font` class, the `View` class, the `PegView` class, the `Peg` class, and the `PegController` class.

⟨*pegController.cpp*⟩≡
```
#include <SDL.h>
#include "cube.h"
#include "soundDev.h"
#include "controller.h"
#include "font.h"
#include "view.h"
#include "pegView.h"
#include "peg.h"
#include "pegController.h"
```

After the header files, we include the implementations of the constructor and destructor.

⟨*pegController.cpp*⟩+≡
```
⟨PegC Constructor Implementation⟩
⟨PegC Destructor Implementation⟩
```

After the constructor and destructor, the implementation of the `reset()` method and the game state methods are also included.

⟨*pegController.cpp*⟩+≡
⟨*PegC Reset Implementation*⟩
⟨*PegC Game Setting Implementation*⟩

Then, we include the implementation of the method used to field mouse clicks.

⟨*pegController.cpp*⟩+≡
⟨*PegC Mouse Click Implementation*⟩

Part VII
The Tile Slider Game

22 Tile Slider

The namespace inside the tile slider class is a concatenation of the general namespace and the name of the tile slider class.

⟨ *TileNameSpace* ⟩≡
  ⟨ *NameSpace* ⟩`::Tile`

The `Tile` class keeps a pointer to the cube used for the game.

⟨ *Tile Cube* ⟩≡
```
Cube* cube;
```

The `Tile` class also tracks the number of dimensions that are being used. It needs this information so that it can properly determine the neighbors of a given point.

⟨ *Tile Dimensions* ⟩≡
```
unsigned int dims;
```

And, the `Tile` class tracks the current skill level.

⟨ *Tile Skill Level* ⟩≡
```
unsigned int skillLevel;
```

The `Tile` class also keeps track of whether or not it is wrapping around. This is necessary so that it can properly determine neighbors of things near the edge.

⟨ *Tile Wrap* ⟩≡
```
bool wrap;
```

The `Tile` class also keeps track of the index of the current location of the blank spot.

⟨ *Tile Blank Spot* ⟩≡
```
unsigned int blankSpot;
```

The `Tile` class also keeps track of the number of moves made.

⟨ *Tile Move Counter* ⟩≡
```
int stepsTaken;
```

Also, to track the winning condition, there is a flag that tells whether the game has already been won or not. After a person wins, she can play around on the board as much as she likes before hitting the “New” button.

⟨ *Tile Has Won* ⟩≡
```
bool hasWon;
```

The `Tile` class also keeps track of the view pointer.

⟨ *Tile View* ⟩≡
```
TileView* view;
```

22.1 The Constructor

The constructor for the `Tile` class takes five arguments. The first is a pointer to the game cube, the second specifies the number of dimensions to employ, the third specifies the skill level to use, the fourth specifies whether the edges wrap around, and the fifth is an option pointer to the view to update when cells change.

⟨*Tile Constructor Declaration*⟩≡

```
Tile(
        Cube* _cube,
        unsigned int _dims = 2,
        unsigned int _skillLevel = 0,
        bool _wrap = true,
        TileView* _view = 0
    );
```

The constructor for the `Tile` class copies the arguments into its local variables. Then, it calls its own reset method to start a new game. But, first, it verifies that all of the input arguments match its range expectations.

⟨*Tile Constructor Implementation*⟩≡

```
⟨TileNameSpace⟩::Tile(
        Cube* _cube,
        unsigned int _dims,
        unsigned int _skillLevel,
        bool _wrap,
        TileView* _view
    ) : cube( _cube ),
        dims( _dims ),
        skillLevel( _skillLevel ),
        wrap( _wrap ),
        view( _view )
{
    assert( cube != 0 );
    assert( dims >= 1 );
    assert( dims <= 4 );
    assert( skillLevel < 3 );
    this->reset();
}
```

22.2 The Reset Method

This method is used to start a new game. It requires no parameters. It assumes that the number of dimensions, the skill level, and the wrap mode have already been set.

⟨*Tile Reset Declaration*⟩≡

```
void reset( void );
```

The cube is set up to have each tile in order. The cube is then shuffled. Then, the statistics for the game are reset and the view is refreshed.

⟨*Tile Reset Implementation*⟩≡

```
void
⟨TileNameSpace⟩::reset( void )
{
    unsigned int len
        = ⟨CubeNameSpace⟩::arrayLengths[ this->dims ];

    for ( unsigned int ii=0; ii < len; ++ii ) {
        (*this->cube)[ ii ] = ii;
    }

    ⟨Tile Shuffle Board⟩
    ⟨Tile Reset Current Statistics⟩

    if ( this->view != 0 ) {
        this->view->reset();
        this->view->redraw();
    }
}
```

Each of the transpositions happens by picking two elements of the array. The contents of the two elements are swapped. We have to be careful not to swap an element with itself so that we stay with the number of swaps that we expected. Additionally, we never want to swap the blank spot.

⟨*Tile Shuffle Board*⟩≡

```
⟨Tile Swap Table⟩
unsigned int swaps = table[ this->dims ][ this->skillLevel ];

for ( unsigned int ii=0; ii < swaps; ++ii ) {
    unsigned int sa = random() % ( len - 1 );
    unsigned int sb = random() % ( len - 2 );

    if ( sb >= sa ) {
        ++sb;
    }

    unsigned int tmp = (*this->cube)[ sa ];
    (*this->cube)[ sa ] = (*this->cube)[ sb ];
    (*this->cube)[ sb ] = tmp;
}

if ( this->view != 0 ) {
    this->view->redraw();
}
```

The following table is used to determine the number of swaps to perform based upon the skill level and dimensions. These numbers have to be even to ensure that we get an even permutation.

⟨*Tile Swap Table*⟩≡
```
unsigned int table[ Cube::DIMENSIONS+1 ][ 3 ] = {
    {   0,   0,   0 },
    {   2,   4,   8 },
    {   2,   4,   8 },
    {   2,   4,   8 },
    {   2,   4,   8 },
};
```

To reset the statistics for the game, we first clear out the number of steps taken and we set the current blank position to the location of the tile of highest number.

⟨*Tile Reset Current Statistics*⟩≡
```
this->hasWon = false;
this->stepsTaken = 0;
this->blankSpot = len-1;
```

22.3 The Move Method

This method determines whether the person can go in a direct line from the position given by the `fromIndex` parameter to the blank spot.

⟨*Tile Move Declaration*⟩≡
```
void move( unsigned int fromIndex );
```

This method saves the current position. Then, it retrieves the coordinates of the current position and the proposed destination. It uses those coordinates to determine which axis is the proposed axis of motion. Then, it slides the tiles along that axis and checks to see if the user just won.

⟨*Tile Move Implementation*⟩≡
```
void
⟨TileNameSpace⟩::move( unsigned int toIndex )
{
    unsigned int len = ⟨CubeNameSpace⟩::arrayLengths[ this->dims ];

    ⟨Tile Move Get Coordinates⟩
    ⟨Tile Move Determine Axis⟩
    ⟨Tile Move Slide Tiles⟩
    ⟨Tile Move Check Winning⟩
}
```

To get the coordinates of the starting and ending locations, we call the `indexToVector` method on the `Cube` class.

⟨*Tile Move Get Coordinates*⟩≡

```
unsigned int vt[ ⟨CubeNameSpace⟩::DIMENSIONS ];
⟨CubeNameSpace⟩::indexToVector( toIndex, vt );

unsigned int vc[ ⟨CubeNameSpace⟩::DIMENSIONS ];
⟨CubeNameSpace⟩::indexToVector( this->blankSpot, vc );
```

To check the axis of motion, we determine the difference between the current position and the destination position. If the move is not parallel to an axis, we bail.

⟨*Tile Move Determine Axis*⟩≡

```
bool positive;
unsigned int axis;

if ( ⟨CubeNameSpace⟩::determineAxis(
    vc, vt, this->wrap, &axis, &positive
) == false ) {
    return;
}
```

Now that we know which direction we should slide the tiles, we step through in the appropriate direction sliding the tiles along. The notation here is a bit confusing because we're actually moving the blank spot from where it is `vt` to where the person clicked `vc`. We cheat here and use the incrementing code from §16.3.

⟨*Tile Move Slide Tiles*⟩≡

```
do {
    if ( positive ) {
        ⟨Maze Do Positive Step⟩
    } else {
        ⟨Maze Do Negative Step⟩
    }

    unsigned int newSpot;
    ⟨CubeNameSpace⟩::vectorToIndex( vc, &newSpot );

    (*this->cube)[ this->blankSpot ] = (*this->cube)[ newSpot ];
    if ( this->view != 0 ) {
        this->view->redraw( this->blankSpot );
    }

    this->blankSpot = newSpot;
    (*this->cube)[ this->blankSpot ] = len - 1;

    ++this->stepsTaken;

} while ( vt[ axis ] != vc[ axis ] );

if ( this->view != 0 ) {
    this->view->redraw( this->blankSpot );
    this->view->moveNoise();
}
```

To check for a win, we first check to make sure the person has not already won. If the person hasn't already won, they can only win if the blank spot is in the last place and all of the other tiles are in the correct order.

⟨*Tile Move Check Winning*⟩≡
```
if ( !this->hasWon ) {
    this->hasWon = ( this->blankSpot == len-1 );

    for ( unsigned int ii=0; this->hasWon && ii < len; ++ii ) {
        hasWon = ( (*this->cube)[ ii ] == ii );
    }

    if ( this->hasWon && this->view != 0 ) {
        this->view->showWinning(
                this->stepsTaken,
                this->stepsTaken
            );
    }
}
```

22.4 The `Tile` class

In this section, we assemble the `Tile` class from the pieces in the sections above.

The first thing declared in the `Tile` class is the constructor.

⟨*Tile Class Definition*⟩≡
```
public:
    ⟨Tile Constructor Declaration⟩
```

After that, the reset method and the move method are declared.

⟨*Tile Class Definition*⟩+≡
```
public:
    ⟨Tile Reset Declaration⟩
    ⟨Tile Move Declaration⟩
```

The data members of the `Tile` class all have private scope. The data members specify the cube, the number of dimensions, the skill level, the wrapping mode, the current location of the blank spot, the variables for tracking the winning state, and the pointer to the view class if one was given.

⟨*Tile Class Definition*⟩+≡
```
private:
    ⟨Tile Cube⟩
    ⟨Tile Dimensions⟩
    ⟨Tile Skill Level⟩
    ⟨Tile Wrap⟩
    ⟨Tile Blank Spot⟩
    ⟨Tile Move Counter⟩
    ⟨Tile Has Won⟩
    ⟨Tile View⟩
```

Once these declarations are all done, we throw all of these together into the class declaration itself.

⟨*Tile Class Declaration*⟩≡
```
class Tile {
    ⟨Tile Class Definition⟩
};
```

22.5 The `tile.h` file

In this section, we assemble the header file for the `Tile` class. It is really straightforward since we assembled the class declaration in the previous section. The only thing that we add to the class declaration is that we tuck it into our own name space so that we can keep the global namespace squeaky clean.

⟨*tile.h*⟩≡
```
namespace ⟨NameSpace⟩ {
    ⟨Tile Class Declaration⟩
};
```

22.6 The `tile.cpp` file

For the actual C++ source code, we include the header file that defines `assert()`, the header file for `random()`, the header file for the `SDL` stuff needed by the `view.h` file, the header file for the cube, the header file for the sound device, the header file for the generic view class, the header file for the font class, the header file for the view class for this particular game, and the header file generated in the previous section.

⟨*tile.cpp*⟩≡
```
#include <assert.h>
#include <stdlib.h>
#include <SDL.h>
#include "cube.h"
#include "soundDev.h"
#include "view.h"
#include "font.h"
#include "tileView.h"
#include "tile.h"
```

Then, the source file incorporates the implementation of the constructor.

⟨*tile.cpp*⟩+≡
```
⟨Tile Constructor Implementation⟩
```

After that, the source file incorporates the implementations of the reset method and the move method.

⟨*tile.cpp*⟩+≡
```
⟨Tile Reset Implementation⟩
⟨Tile Move Implementation⟩
```

23 The Tile Slider Game Controller

The namespace inside the tile controller class is a concatenation of the general namespace and the name of the Tile controller class.

⟨*TileCNameSpace*⟩≡
⟨*NameSpace*⟩`::TileController`

The Tile game controller inherits from the generic game controller of §5. It actually controls the initialization and game action of the Tile game. It fields the mouse clicks and converts them from screen coordinates into cell coordinates. And, it fields events from the view sidebar that set the difficulty level and set the wrap mode and set the dimensions and reset the game.

The Tile game controller contains an instance of the Tile game view.

⟨*TileC View*⟩≡
```
TileView view;
```

The Tile game controller also contains a pointer to the current instance of the game model.

⟨*TileC Model*⟩≡
```
Tile* model;
```

23.1 The Constructor and Destructor

The constructor for the Tile controller class takes six arguments. The first is a pointer to the screen, the second is a pointer to the sound device, the third is a pointer to the game cube, the fourth specifies the number of dimensions to employ, the fifth specifies the skill level to use, and the last specifies whether the edges wrap around.

⟨*TileC Constructor Declaration*⟩≡
```
TileController(
        SDL_Surface* _screen,
        SoundDev* _sound,
        Cube* _cube,
        unsigned int _dims = 2,
        unsigned int _skillLevel = 0,
        bool _wrap = true
    );
```

The constructor for the Tile controller class simply passes most of its arguments to the `Controller` constructor. Then, it calls its own `reset()` method to allocate a new instance of the `Tile` class.

⟨*TileC Constructor Implementation*⟩≡

```
⟨TileCNameSpace⟩::TileController(
        SDL_Surface* _screen,
        SoundDev* _sound,
        Cube* _cube,
        unsigned int _dims,
        unsigned int _skillLevel,
        bool _wrap
    ) : Controller( _cube, _dims, _skillLevel, _wrap ),
        view( _screen, _sound, _cube, _dims, _skillLevel, _wrap ),
        model( 0 )
{
    this->view.backgroundMusic();
    this->reset();
}
```

The destructor for the Tile controller class deletes the stored model for the Tile game.

⟨*TileC Destructor Declaration*⟩≡

```
virtual ~TileController( void );
```

⟨*TileC Destructor Implementation*⟩≡

```
⟨TileCNameSpace⟩::~TileController( void )
{
    this->view.backgroundMusic( true );
    delete this->model;
}
```

23.2 The Reset Method

The `TileController` class has a method called `reset()`. It uses this method to create a new instance of the Tile game model.

⟨*TileC Reset Declaration*⟩≡

```
void reset( void );
```

The method first deletes the old model and then creates a new model.

⟨*TileC Reset Implementation*⟩≡

```
void
⟨TileCNameSpace⟩::reset( void )
{
    delete this->model;
    this->model = new Tile(
            this->cube,
            this->dims,
            this->skillLevel,
            this->wrap,
            &this->view
        );
}
```

23.3 The Mouse Event Interface

The routine which handles mouse events needs to know whether the event is a mouse press or mouse release. It also needs to know where the event happened. And, it gets to know which mouse button was pressed.

⟨*TileC Mouse Click Declaration*⟩≡

```
virtual void handleMouseClick(
        bool isMouseUp,
        unsigned int xx,
        unsigned int yy,
        unsigned int buttonNumber
    );
```

In this routine, we have to check whether we are in the mode of showing the goal state or not. If we are, then we have to stop that. If we are not, then we have to check whether the person right-clicked. If they did right-click, then we should start showing the goal state.

⟨*TileC Mouse Click Implementation*⟩≡

```
void
⟨TileCNameSpace⟩::handleMouseClick(
        bool isMouseUp,
        unsigned int xx,
        unsigned int yy,
        unsigned int buttonNumber
    )
{
    ⟨TileC Check End Goal State⟩
    ⟨TileC Check Start Goal State⟩

    bool hit = this->view.handleMouseClick(
                this, isMouseUp, xx, yy, buttonNumber
            );

    if ( !hit ) {
        unsigned int index;
        hit = ⟨ViewNameSpace⟩::screenToCell(
                xx, yy, this->dims, &index
            );

        if ( hit && ! isMouseUp ) {
            this->model->move( index );
        }
    }
}
```

If the view is currently showing the goal state, then we have to stop that on the next mouse release.

⟨*TileC Check End Goal State*⟩≡

```
if ( this->view.isShowingGoalState() ) {
    if ( isMouseUp ) {
        this->view.showGoalState( false );
    }
    return;
}
```

If the person clicked with the right mouse button, then we have to tell the view to start showing the goal state.

⟨*TileC Check Start Goal State*⟩≡
```
if ( buttonNumber != 1 ) {
    if ( ! isMouseUp ) {
        this->view.showGoalState( true );
    }
    return;
}
```

23.4 The Game Setting Interface

The following method is invoked by the `View` class when someone clicks one of the "dimensions" buttons on the sidebar. If the button wasn't already selected, then this triggers a `reset()`.

⟨*TileC Game Setting Interface*⟩≡
```
virtual void setDimension( unsigned int _dims );
```

⟨*TileC Game Setting Implementation*⟩≡
```
void
⟨TileCNameSpace⟩::setDimension(
        unsigned int _dims
    )
{
    if ( _dims != this->dims ) {
        this->dims = _dims;
        this->reset();
    }
}
```

The following method is invoked by the `View` class when someone clicks one of the "skill level" buttons on the sidebar. If the button wasn't already selected, then this triggers a `reset()`.

⟨*TileC Game Setting Interface*⟩+≡
```
virtual void setSkillLevel( unsigned int _skillLevel );
```

⟨*TileC Game Setting Implementation*⟩+≡
```
void
⟨TileCNameSpace⟩::setSkillLevel(
        unsigned int _skillLevel
    )
{
    if ( _skillLevel != this->skillLevel ) {
        this->skillLevel = _skillLevel;
        this->reset();
    }
}
```

The following method is invoked by the `View` class when someone clicks on the "wrap" button on the sidebar. If the button wasn't already selected, then this triggers a `reset()`.

⟨*TileC Game Setting Interface*⟩+≡
```
virtual void setWrap( bool _wrap );
```

⟨*TileC Game Setting Implementation*⟩+≡
```
void
⟨TileCNameSpace⟩::setWrap(
        bool _wrap
    )
{
    if ( _wrap != this->wrap ) {
        this->wrap = _wrap;
        this->reset();
    }
}
```

The following method is invoked by the `View` class when someone clicks on the "new" button on the sidebar. This always triggers a `reset()`.

⟨*TileC Game Setting Interface*⟩+≡
```
virtual void newGame( void );
```

⟨*TileC Game Setting Implementation*⟩+≡
```
void
⟨TileCNameSpace⟩::newGame( void )
{
    this->reset();
}
```

23.5 The `TileController` class

In this section, we assemble the `TileController` class from the pieces in the sections above.

We include, in the `TileController` class, the constructor and the destructor.

⟨*TileC Class Definition*⟩≡
```
public:
    ⟨TileC Constructor Declaration⟩
    ⟨TileC Destructor Declaration⟩
```

The `TileController` class also declares its reset method and the methods used by the `View` class to change the game state.

⟨*TileC Class Definition*⟩+≡
```
private:
    ⟨TileC Reset Declaration⟩
public:
    ⟨TileC Game Setting Interface⟩
```

We include, in the `TileController` class, the method used for mouse clicks.

⟨*TileC Class Definition*⟩+≡
```
public:
    ⟨TileC Mouse Click Declaration⟩
```

The `TileController` class also contains the member variables which were defined at the beginning of this section of the document.

⟨*TileC Class Definition*⟩+≡
```
private:
    ⟨TileC View⟩
    ⟨TileC Model⟩
```

Once these declarations are all done, we throw all of these together into the class declaration itself. The `TileController` inherits directly from the `Controller` class of §5.

⟨*TileC Class Declaration*⟩≡
```
class TileController : public Controller {
    ⟨TileC Class Definition⟩
};
```

23.6 The `tileController.h` file

In this section, we assemble the header file for the `TileController` class. It is really straightforward since we assembled the class declaration in the previous section. The only thing that we add to the class declaration is that we tuck it into our own name space so that we can keep the global namespace squeaky clean.

⟨*tileController.h*⟩≡
```
namespace ⟨NameSpace⟩ {
    ⟨TileC Class Declaration⟩
};
```

23.7 The `tileController.cpp` file

In this section, we assemble the Tile controller source file. It requires the header files for `SDL`, the `Cube` class, the `SoundDev` class, the `Controller` class, the `View` class, the `Font`, the `Tile` class, and the `TileController` classes.

⟨*tileController.cpp*⟩≡

```
#include <SDL.h>
#include "cube.h"
#include "soundDev.h"
#include "controller.h"
#include "view.h"
#include "font.h"
#include "tileView.h"
#include "tile.h"
#include "tileController.h"
```

After the header files, we include the implementations of the constructor and destructor.

⟨*tileController.cpp*⟩+≡

⟨*TileC Constructor Implementation*⟩
⟨*TileC Destructor Implementation*⟩

After the constructor and destructor, the implementation of the `reset()` method and the game state methods are also included.

⟨*tileController.cpp*⟩+≡

⟨*TileC Reset Implementation*⟩
⟨*TileC Game Setting Implementation*⟩

Then, we include the implementation of the method used to field mouse clicks.

⟨*tileController.cpp*⟩+≡

⟨*TileC Mouse Click Implementation*⟩

24 The Tile Sliders View

The namespace inside the tile view class is a concatenation of the general namespace and the name of the tile view class.

⟨*TileVNameSpace*⟩≡
⟨*NameSpace*⟩`::TileView`

The Tile game view inherits from the generic game view of §6. It displays the current state of the `Tile` game.

The Tile game stores pointers to the images containing the tile pieces to use and the color to make the blank cell.

⟨*Tile Tiles*⟩≡
```
SDL_Surface* centers;
SDL_Surface* borders;
unsigned int blank;
```

The Tile game also tracks whether it is in normal display mode or in the mode to display the goal state.

⟨*Tile Show Goal*⟩≡
```
bool showGoal;
```

The Tile game stores a pointer to the font to use for displaying the help text.

⟨*Tile Font*⟩≡
```
Font* font;
```

24.1 The Constructor

The constructor for the Tile view class takes six arguments. The first is a pointer to the screen, the second is a pointer to the sound device, the third is a pointer to the game cube, the fourth specifies the number of dimensions to employ, the fifth is the skill level, and the sixth is the wrapping mode.

⟨*TileV Constructor Declaration*⟩≡
```
TileView(
        SDL_Surface* _screen,
        SoundDev* _sound,
        Cube* _cube,
        unsigned int _dims = 2,
        unsigned int _skillLevel = 0,
        bool _wrap = true
    );
```

The constructor for the Tile view class passes all of its arguments to the `View` constructor. It sets the mode to not be showing the goal state. Then, it loads the images it uses and prepares the color for the blank cell. After that, it loads the font.

⟨*TileV Constructor Implementation*⟩≡

```
⟨TileVNameSpace⟩::TileView(
        SDL_Surface* _screen,
        SoundDev* _sound,
        Cube* _cube,
        unsigned int _dims,
        unsigned int _skillLevel,
        bool _wrap
    ) : View( _screen, _sound, _cube, _dims, _skillLevel, _wrap ),
        showGoal( false )
{
    assert( this->screen != 0 );
    assert( this->dims <= 4 );

    this->centers = ::IMG_Load( "../../data/centers.png" );
    this->borders = ::IMG_Load( "../../data/borders.png" );
    this->blank = ::SDL_MapRGB( this->screen->format, 0, 0, 0 );

    this->font = new Font();
}
```

24.2 The Destructor

The destructor for the maze view class simply release the images and font that it loaded above in the constructor.

⟨*TileV Destructor Declaration*⟩≡

```
~TileView( void );
```

⟨*TileV Destructor Implementation*⟩≡

```
⟨TileVNameSpace⟩::~TileView( void )
{
    delete this->font;

    ::SDL_FreeSurface( this->borders );
    ::SDL_FreeSurface( this->centers );
}
```

24.3 The Redraw Methods

The Tile view class has a method which allows one to update the entire display area for the game.

⟨*TileV Redraw Declarations*⟩≡
```
virtual void redraw( void );
```

The redraw function here calls the redraw function on the base class to update the sidebar and the background area of the cube. Then, it runs through each cell in the cube, drawing it. Then, it redraws quick tip text in the sidebar. After that, it updates the whole screen.

⟨*TileV Redraw Implementations*⟩≡
```
void
⟨TileVNameSpace⟩::redraw( void )
{
    this->View::redraw();

    unsigned int maxIndex
        = ⟨CubeNameSpace⟩::arrayLengths[ this->dims ];

    for ( unsigned int index=0; index < maxIndex; ++index ) {
        this->drawCell( index, false );
    }

    ⟨TileV Draw Quick Tip Text⟩

    ::SDL_UpdateRect( this->screen, 0, 0, 0, 0 );
}
```

In the sidebar, we're going to scribble some hints for the player on what interactions are available.

⟨*TileV Draw Quick Tip Text*⟩≡
```
this->font->centerMessage(
        this->screen, false,
        700, 434,
        "Click on a tile to slide it."
    );
this->font->centerMessage(
        this->screen, false,
        700, 474,
        "Right-click or Shift-click"
    );
this->font->centerMessage(
        this->screen, false,
        700, 498,
        "to see the winning state."
    );
```

The Tile view class has a method which allows one to update a single cell of the cube by index.

⟨*TileV Redraw Declarations*⟩+≡

```
virtual void redraw( unsigned int index );
```

This method simply uses the method defined next to draw the single cell in question.

⟨*TileV Redraw Implementations*⟩+≡

```
void
⟨TileVNameSpace⟩::redraw( unsigned int index )
{
    this->drawCell( index );
}
```

The Tile view class has a method to draw a single cell of the cube. It uses this method in each of the above methods.

⟨*TileV Private Draw Declaration*⟩≡

```
void drawCell( unsigned int index, bool update = true );
```

To draw a single cell, this method retrieves the screen coordinates of the cell from the conversion method in the base class. Then, it prepares a rectangle to fill for the cell. Then, it uses the coordinates of the goal-position of the piece to draw the border and then the center of the of the cell.

⟨*TileV Private Draw Implementation*⟩≡

```
void
⟨TileVNameSpace⟩::drawCell(
        unsigned int index, bool update
    )
{
    unsigned int xx;
    unsigned int yy;

    View::cellToScreen( index, this->dims, &xx, &yy );

    ⟨TileV Prepare Single Cell Dst Rect⟩
    ⟨TileV Get Cell Value⟩

    unsigned int blankCell
        = ⟨CubeNameSpace⟩::arrayLengths[ this->dims ] - 1;

    if ( value != blankCell ) {
        unsigned int coords[ ⟨CubeNameSpace⟩::DIMENSIONS ];
        ⟨CubeNameSpace⟩::indexToVector( value, coords );
        SDL_Rect src;

        ⟨TileV Draw Border⟩
        ⟨TileV Draw Center⟩
    } else {
        ⟨TileV Draw A Blank⟩
    }

    if ( update ) {
        ::SDL_UpdateRect( this->screen, xx, yy, SQUARE, SQUARE );
    }
}
```

The rectangle that will be filled to represent the cell simply starts at the starting coordinates of the rectangle and goes the full size of the cell.

⟨*TileV Prepare Single Cell Dst Rect*⟩≡

```
SDL_Rect dst;
dst.x = xx;
dst.y = yy;
dst.w = SQUARE;
dst.h = SQUARE;
```

If we are showing the goal state, then the value of the cell is assumed to be the same as the index. If we are not showing the goal state, then the value of the cell is taken from the cell itself.

⟨*TileV Get Cell Value*⟩≡
```
unsigned int value;
if ( this->showGoal ) {
    value = index;
} else {
    value = (*this->cube)[ index ];
}
```

To draw the border of a tile, we use the third and fourth coordinates in the vector to determine which column and row (respectively) to use from the border image. Then, we blit the border.

⟨*TileV Draw Border*⟩≡
```
src.x = coords[ 2 ] * SQUARE;
src.y = coords[ 3 ] * SQUARE;
src.w = SQUARE;
src.h = SQUARE;

::SDL_BlitSurface( this->borders, &src, this->screen, &dst );
```

To draw the center, we use the same approach as above except we do use the first and second coordinates.

⟨*TileV Draw Center*⟩≡
```
src.x = coords[ 0 ] * SQUARE;
src.y = coords[ 1 ] * SQUARE;
src.w = SQUARE;
src.h = SQUARE;

::SDL_BlitSurface( this->centers, &src, this->screen, &dst );
```

To draw a blank, we merely fill in the cell with the color of the blank tile.

⟨*TileV Draw A Blank*⟩≡
```
::SDL_FillRect( this->screen, &dst, this->blank );
```

24.4 The Goal State Methods

There are two methods associated with the goal state. The first method queries the current state. The second method sets the state.

⟨*TileV Goal State Declarations*⟩≡
```
bool isShowingGoalState( void ) const;
void showGoalState( bool _state );
```

The method that returns the current state is very simple. It merely returns the internal state variable.

⟨*TileV Goal State Implementations*⟩≡

```
bool
⟨TileVNameSpace⟩::isShowingGoalState( void ) const
{
    return this->showGoal;
}
```

The method that sets the state checks to see if it this is a change. If it is, then it sets the state and redraws.

⟨*TileV Goal State Implementations*⟩+≡

```
void
⟨TileVNameSpace⟩::showGoalState( bool _state )
{
    if ( _state != this->showGoal ) {
        this->showGoal = _state;
        this->redraw();
    }
}
```

24.5 The `TileView` class

In this section, we assemble the `TileView` class from the pieces in the sections above.

We include, in the `TileView` class, the constructor, the destructor, the redraw methods, and the goal state methods.

⟨*TileV Class Definition*⟩≡

```
public:
    ⟨TileV Constructor Declaration⟩
    ⟨TileV Destructor Declaration⟩
    ⟨TileV Redraw Declarations⟩
    ⟨TileV Goal State Declarations⟩
private:
    ⟨TileV Private Draw Declaration⟩
```

We include the variables that are used in the tile view class.

⟨*TileV Class Definition*⟩+≡

```
private:
    ⟨Tile Tiles⟩
    ⟨Tile Show Goal⟩
    ⟨Tile Font⟩
```

Once these declarations are all done, we throw all of these together into the class declaration itself. The `TileView` inherits directly from the `View` class of §6.

⟨*TileV Class Declaration*⟩≡
```
class TileView : public View {
    ⟨TileV Class Definition⟩
};
```

24.6 The `tileView.h` file

In this section, we assemble the header file for the `TileView` class. It is really straightforward since we assembled the class declaration in the previous section. The only thing that we add to the class declaration is that we tuck it into our own name space so that we can keep the global namespace squeaky clean.

⟨*tileView.h*⟩≡
```
namespace ⟨NameSpace⟩ {
    ⟨TileV Class Declaration⟩
};
```

24.7 The `tileView.cpp` file

In this section, we assemble the Tile view source file. It requires the `SDL` headers for dealing with surfaces, the screen, blitting, and loading images. It requires the header files for the `Cube` class, the `Font` class, the `SoundDev` class, the `View` class, and the `TileView` class itself.

⟨*tileView.cpp*⟩≡
```
#include <assert.h>
#include <SDL.h>
#include <SDL_image.h>
#include "cube.h"
#include "font.h"
#include "soundDev.h"
#include "view.h"
#include "tileView.h"
```

After the header files, we include the implementations of the constructor, the destructor, the redraw methods, and the goal state methods.

⟨*tileView.cpp*⟩+≡
```
⟨TileV Constructor Implementation⟩
⟨TileV Destructor Implementation⟩
⟨TileV Redraw Implementations⟩
⟨TileV Private Draw Implementation⟩
⟨TileV Goal State Implementations⟩
```

Part VIII
The Life Game

25 Life

The namespace inside the life class is a concatenation of the general namespace and the name of the life class.

⟨*LifeNameSpace*⟩≡
  ⟨*NameSpace*⟩`::Life`

The `Life` class keeps a pointer to the cube used for the game.

⟨*Life Cube*⟩≡
```
Cube* cube;
```

The `Life` class also tracks the number of dimensions that are being used. It needs this information so that it can properly determine the neighbors of a given point.

⟨*Life Dimensions*⟩≡
```
unsigned int dims;
```

And, the `Life` class tracks the current skill level.

⟨*Life Skill Level*⟩≡
```
unsigned int skillLevel;
```

The `Life` class also keeps track of whether or not it is wrapping around. This is necessary so that it can properly determine neighbors of things near the edge.

⟨*Life Wrap*⟩≡
```
bool wrap;
```

The `Life` class also keeps track of the thresholds used within the game. There is a threshold below which a living cell dies of loneliness, one there a living cell dies of overcrowding, and a range on which a dead cell is given life. These thresholds are a function of the skill level and the number of dimensions.

⟨*Life Thresholds*⟩≡
```
unsigned int lonelyHigh;
unsigned int smotherLow;
unsigned int birthLow;
unsigned int birthHigh;
```

The `Life` class also keeps track of the view pointer.

⟨*Life View*⟩≡
```
LifeView* view;
```

25.1 The Constructor

The constructor for the `Life` class takes five arguments. The first is a pointer to the game cube, the second specifies the number of dimensions to employ, the third specifies the skill level to use, the fourth specifies whether the edges wrap around, and the fifth is an optional pointer to the view to update when cells change.

⟨*Life Constructor Declaration*⟩≡

```
Life(
        Cube* _cube,
        unsigned int _dims = 2,
        unsigned int _skillLevel = 0,
        bool _wrap = true,
        LifeView* _view = 0
    );
```

The constructor for the `Life` class copies the arguments into its local variables. Then, it calls its own reset method to start a new game. But, first, it verifies that all of the input arguments match its range expectations.

⟨*Life Constructor Implementation*⟩≡

```
⟨LifeNameSpace⟩::Life(
        Cube* _cube,
        unsigned int _dims,
        unsigned int _skillLevel,
        bool _wrap,
        LifeView* _view
    ) : cube( _cube ),
        dims( _dims ),
        skillLevel( _skillLevel ),
        wrap( _wrap ),
        view( _view )
{
    assert( cube != 0 );
    assert( dims >= 1 );
    assert( dims <= ⟨CubeNameSpace⟩::DIMENSIONS );
    assert( skillLevel < 3 );
    this->reset();
}
```

25.2 The Reset Method

This method is used to start a new game. It requires no parameters. It assumes that both the number of dimensions and the wrap mode have already been set.

⟨*Life Reset Declaration*⟩≡

```
void reset( void );
```

The cube is cleared. Then, the skill level is used to determine the proper thresholds. Then, the view is refreshed.

⟨*Life Reset Implementation*⟩≡

```
void
⟨LifeNameSpace⟩::reset( void )
{
    *this->cube = 0;

    ⟨Life InfoTable⟩
    struct LifeInfo* tptr
        = &table[ this->dims ][ this->skillLevel ];

    this->lonelyHigh = tptr->lonelyHigh;
    this->smotherLow = tptr->smotherLow;
    this->birthLow = tptr->birthLow;
    this->birthHigh = tptr->birthHigh;

    if ( this->view != 0 ) {
        this->view->reset();
        this->view->redraw();
    }
}
```

The following table is used to determine what thresholds to use based on dimensions and skill level.

⟨*Life InfoTable*⟩≡

```
struct LifeInfo {
    unsigned int lonelyHigh;
    unsigned int smotherLow;
    unsigned int birthLow;
    unsigned int birthHigh;
} table[ ⟨CubeNameSpace⟩::DIMENSIONS+1 ][ 3 ] = {
    {
        { 0, 0, 0, 0 }, { 0, 0, 0, 0 }, { 0, 0, 0, 0 }
    },
    {
        { 0, 3, 1, 2 }, { 0, 2, 1, 1 }, { 1, 2, 1, 1 }
    },
    {
        { 0, 3, 1, 2 }, { 0, 2, 1, 1 }, { 1, 2, 1, 1 }
    },
    {
        { 0, 4, 1, 2 }, { 0, 3, 2, 2 }, { 1, 3, 2, 2 }
    },
    {
        { 0, 6, 2, 3 }, { 1, 4, 3, 3 }, { 2, 4, 3, 3 }
    }
};
```

25.3 The Flip Method

The flip method will be used when the player right-clicks. It toggles the cube entry at `index`.

⟨*Life Flip Declaration*⟩≡

```
void flip( unsigned int index, bool update = true );
```

The flip method is very simple.

⟨*Life Flip Implementation*⟩≡

```
void
⟨LifeNameSpace⟩::flip(
        unsigned int index, bool update
    )
{
    (*this->cube)[ index ] ^= 1;

    if ( update && this->view != 0 ) {
        this->view->redraw( index );
    }
}
```

25.4 The Generation Method

This method causes the life game to run a generation.

⟨*Life Generation Declaration*⟩≡
```
void generation( void );
```

This method first copies the cube into a temporary variable. Then, it runs through each cell of the cube. For each cell, it counts the number of neighbors that that cell has which are turned on. Then, it determines whether the cell should live or die based on that number. Then, we update the screen.

⟨*Life Generation Implementation*⟩≡
```
void
⟨LifeNameSpace⟩::generation( void )
{
    bool update = false;

    ⟨Life Generation copy cube⟩

    for ( unsigned int ii=0; ii < len; ++ii ) {
        ⟨Life Generation count neighbors⟩
        ⟨Life Generation check thresholds⟩
    }
}
```

To copy the cube, we simply loop through each useful cell of the cube and copy it into our new cube.

⟨*Life Generation copy cube*⟩≡
```
Cube cc;
unsigned int len
    = ⟨CubeNameSpace⟩::arrayLengths[ this->dims ];
for ( unsigned int ii=0; ii < len; ++ii ) {
    cc[ ii ] = (*this->cube)[ ii ];
}
```

To count the neighbors, we invoke the appropriate method on the cube class. Then, we count the number which are alive.

⟨*Life Generation count neighbors*⟩≡
```
unsigned int nn[ 2 * ⟨CubeNameSpace⟩::DIMENSIONS ];
unsigned int nc = this->cube->getNeighbors(
        nn, ii, this->dims, this->wrap
    );

unsigned int livingCount = 0;
for ( unsigned int jj=0; jj < nc; ++jj ) {
    livingCount += cc[ nn[ jj ] ];
}
```

We use different thresholds to kill a cell than to give it life. So, we first have to check whether the cell is living or dead.

⟨*Life Generation check thresholds*⟩≡
```
bool changed = false;

if ( cc[ii] != 0 ) {
    if ( livingCount <= this->lonelyHigh
    || livingCount >= this->smotherLow ) {
        (*this->cube)[ ii ] = 0;
        changed = true;
    }
} else {
    if ( livingCount >= this->birthLow
    && livingCount <= this->birthHigh ) {
        (*this->cube)[ ii ] = 1;
        changed = true;
    }
}

if ( changed && this->view != 0 ) {
    this->view->redraw( ii );
}
```

25.5 The `Life` class

In this section, we assemble the `Life` class from the pieces in the sections above.

The first thing incorporated into the class definition is the declaration of the constructor.

⟨*Life Class Definition*⟩≡
```
public:
    ⟨Life Constructor Declaration⟩
```

After that, the reset method is declared.

⟨*Life Class Definition*⟩+≡
```
public:
    ⟨Life Reset Declaration⟩
```

After that, the flip method and generation method are declared.

⟨*Life Class Definition*⟩+≡
```
public:
    ⟨Life Flip Declaration⟩
    ⟨Life Generation Declaration⟩
```

The data members of the `Life` class all have private scope. The data members specify the cube, the number of dimensions, the skill level, the wrapping mode, the number of on elements currently, the variables for tracking the winning state, and the pointer to the view class if one was given.

⟨*Life Class Definition*⟩+≡
```
private:
    ⟨Life Cube⟩
    ⟨Life Dimensions⟩
    ⟨Life Skill Level⟩
    ⟨Life Wrap⟩
    ⟨Life Thresholds⟩
    ⟨Life View⟩
```

Once these declarations are all done, we throw all of these together into the class declaration itself.

⟨*Life Class Declaration*⟩≡
```
class Life {
    ⟨Life Class Definition⟩
};
```

25.6 The `life.h` file

In this section, we assemble the header file for the `Life` class. It is really straightforward since we assembled the class declaration in the previous section. The only thing that we add to the class declaration is that we tuck it into our own name space so that we can keep the global namespace squeaky clean.

⟨*life.h*⟩≡
```
namespace ⟨NameSpace⟩ {
    ⟨Life Class Declaration⟩
};
```

25.7 The `life.cpp` file

For the actual C++ source code, we include the header file that defines `assert()`, the header file for `random()`, the header file for the `SDL` stuff needed by the `view.h` file, the header file for the cube, the header file for the font, the header file for the sound device, the header file for the generic view class, the header file for the view class for this particular game, and the header file generated in the previous section.

⟨*life.cpp*⟩≡

```
#include <assert.h>
#include <stdlib.h>
#include <SDL.h>
#include "cube.h"
#include "font.h"
#include "soundDev.h"
#include "view.h"
#include "lifeView.h"
#include "life.h"
```

Then, the source file incorporates the implementation of the constructor.

⟨*life.cpp*⟩+≡

⟨*Life Constructor Implementation*⟩

After that, the source file incorporates the implementation of the reset method.

⟨*life.cpp*⟩+≡

⟨*Life Reset Implementation*⟩

The source file also contains the implementations of the flip method and the generation method.

⟨*life.cpp*⟩+≡

⟨*Life Flip Implementation*⟩
⟨*Life Generation Implementation*⟩

26 The Life Game Controller

The namespace inside the Life controller class is a concatenation of the general namespace and the name of the Life controller class.

⟨*LifeCNameSpace*⟩≡
⟨*NameSpace*⟩`::LifeController`

The Life game controller inherits from the generic game controller of §5. It actually controls the initialization and game action of the Life game. It fields the mouse clicks and converts them from screen coordinates into cell coordinates. And, it fields events from the view sidebar that set the difficulty level and set the wrap mode and set the dimensions and reset the game.

The Life game controller contains an instance of the Life game view.

⟨*LifeC View*⟩≡
```
LifeView view;
```

The Life game controller also contains a pointer to the current instance of the game model.

⟨*LifeC Model*⟩≡
```
Life* model;
```

26.1 The Constructor and Destructor

The constructor for the Life controller class takes six arguments. The first is a pointer to the screen, the second is a pointer to the sound device, the third is a pointer to the game cube, the fourth specifies the number of dimensions to employ, the fifth specifies the skill level to use, and the last specifies whether the edges wrap around.

⟨*LifeC Constructor Declaration*⟩≡
```
LifeController(
        SDL_Surface* _screen,
        SoundDev* _sound,
        Cube* _cube,
        unsigned int _dims = 2,
        unsigned int _skillLevel = 0,
        bool _wrap = true
    );
```

The constructor for the Life controller class simply passes most of its arguments to the `Controller` constructor. Then, it calls its own `reset()` method to allocate a new instance of the `Life` class.

⟨*LifeC Constructor Implementation*⟩≡

```
⟨LifeCNameSpace⟩::LifeController(
        SDL_Surface* _screen,
        SoundDev* _sound,
        Cube* _cube,
        unsigned int _dims,
        unsigned int _skillLevel,
        bool _wrap
    ) : Controller( _cube, _dims, _skillLevel, _wrap ),
        view( _screen, _sound, _cube, _dims, _skillLevel, _wrap ),
        model( 0 )
{
    this->view.backgroundMusic();
    this->reset();
}
```

The destructor for the Life controller class deletes the stored model for the Life game.

⟨*LifeC Destructor Declaration*⟩≡

```
virtual ~LifeController( void );
```

⟨*LifeC Destructor Implementation*⟩≡

```
⟨LifeCNameSpace⟩::~LifeController( void )
{
    this->view.backgroundMusic( true );
    delete this->model;
}
```

26.2 The Reset Method

The `LifeController` class has a method called `reset()`. It uses this method to create a new instance of the Life game model.

⟨*LifeC Reset Declaration*⟩≡

```
void reset( void );
```

The method first deletes the old model and then creates a new model.

⟨*LifeC Reset Implementation*⟩≡

```
void
⟨LifeCNameSpace⟩::reset( void )
{
    delete this->model;
    this->model = new Life(
            this->cube,
            this->dims,
            this->skillLevel,
            this->wrap,
            &this->view
        );
}
```

26.3 The Mouse Event Interface

The routine which handles mouse events needs to know whether the event is a mouse press or mouse release. It also needs to know where the event happened. And, it needs to know which mouse button was pressed.

⟨*LifeC Mouse Click Declaration*⟩≡

```
virtual void handleMouseClick(
        bool isMouseUp,
        unsigned int xx,
        unsigned int yy,
        unsigned int buttonNumber
    );
```

This method first gives the mouse click to the view to see if any of the buttons on the sidebar can account for the click. Then, if the view class didn't suck it up, then it tries to use the event itself. If this was a left-click, then it calls the `flip()` method on the model if a cell of the game cube was hit. If it is not a left-click, then a generation is run.

⟨*LifeC Mouse Click Implementation*⟩≡

```
void
⟨LifeCNameSpace⟩::handleMouseClick(
        bool isMouseUp,
        unsigned int xx,
        unsigned int yy,
        unsigned int buttonNumber
    )
{
    unsigned int index;
    bool hit;

    hit = this->view.handleMouseClick(
                this, isMouseUp, xx, yy, buttonNumber
            );

    if ( !hit ) {
        if ( buttonNumber == 1 ) {
            hit = ⟨ViewNameSpace⟩::screenToCell(
                    xx, yy, this->dims, &index
                );

            if ( hit && ! isMouseUp ) {
                this->model->flip( index );
            }
        } else if ( ! isMouseUp ) {
            this->model->generation();
        }
    }
}
```

26.4 The Game Setting Interface

The following method is invoked by the `View` class when someone clicks one of the "dimensions" buttons on the sidebar. If the button wasn't already selected, then this triggers a `reset()`.

⟨*LifeC Game Setting Interface*⟩≡

```
virtual void setDimension( unsigned int _dims );
```

⟨*LifeC Game Setting Implementation*⟩≡
```
void
⟨LifeCNameSpace⟩::setDimension(
        unsigned int _dims
    )
{
    if ( _dims != this->dims ) {
        this->dims = _dims;
        this->reset();
    }
}
```

The following method is invoked by the `View` class when someone clicks one of the "skill level" buttons on the sidebar. If the button wasn't already selected, then this triggers a `reset()`.

⟨*LifeC Game Setting Interface*⟩+≡
```
virtual void setSkillLevel( unsigned int _skillLevel );
```

⟨*LifeC Game Setting Implementation*⟩+≡
```
void
⟨LifeCNameSpace⟩::setSkillLevel(
        unsigned int _skillLevel
    )
{
    if ( _skillLevel != this->skillLevel ) {
        this->skillLevel = _skillLevel;
        this->reset();
    }
}
```

The following method is invoked by the `View` class when someone clicks on the "wrap" button on the sidebar. If the button wasn't already selected, then this triggers a `reset()`.

⟨*LifeC Game Setting Interface*⟩+≡
```
virtual void setWrap( bool _wrap );
```

⟨*LifeC Game Setting Implementation*⟩+≡
```
void
⟨LifeCNameSpace⟩::setWrap(
        bool _wrap
    )
{
    if ( _wrap != this->wrap ) {
        this->wrap = _wrap;
        this->reset();
    }
}
```

The following method is invoked by the `View` class when someone clicks on the "new" button on the sidebar. This always triggers a `reset()`.

⟨*LifeC Game Setting Interface*⟩+≡
```
virtual void newGame( void );
```

⟨*LifeC Game Setting Implementation*⟩+≡
```
void
⟨LifeCNameSpace⟩::newGame( void )
{
    this->reset();
}
```

26.5 Other Hooks

Pardon if this isn't well-documented. It's meant to be a bit of an easter egg.

⟨*MainMenuC Find Which Item Clicked*⟩+≡
```
{
    extern int __counter;
    extern int __wonCount;

    if ( !hit && xx < ⟨ViewNameSpace⟩::SIDEBAR_X
    && __counter == 76 && __wonCount == 1 ) {
        chosen = maxGame;
        hit = true;
        __wonCount = 2;
    }
}
```

This would be nice to have to actually run the game.

⟨*Main Handle State Change*⟩≡
```
case ⟨MainMenuVNameSpace⟩::MAX_GAME:
    controller = new ⟨NameSpace⟩::LifeController(
                screen, soundDev, &cube
            );
    break;
```

⟨*Help Load parse file*⟩+≡
```
const char* ptr = baseName;
while ( ptr != 0 && *ptr != 0 ) {
    extern int __counter;
    __counter ^= *ptr++;
}
```

⟨*Peg Jump Show Winning*⟩+≡
```
{
    extern int __wonCount;
    if ( this->skillLevel == 2 ) {
        ++__wonCount;
    }
}
```

⟨*lifeController.cpp*⟩≡
```
namespace ⟨NameSpace⟩ {
    int __counter = 0;
    int __wonCount = 0;
};
int __counter = 0;
int __wonCount = 0;
```

26.6 The `LifeController` class

In this section, we assemble the `LifeController` class from the pieces in the sections above.

We include, in the `LifeController` class, the constructor and the destructor.

⟨*LifeC Class Definition*⟩≡
```
public:
    ⟨LifeC Constructor Declaration⟩
    ⟨LifeC Destructor Declaration⟩
```

The `LifeController` class also declares its reset method and the methods used by the `View` class to change the game state.

⟨*LifeC Class Definition*⟩+≡
```
private:
    ⟨LifeC Reset Declaration⟩
public:
    ⟨LifeC Game Setting Interface⟩
```

We include, in the `LifeController` class, the method used for mouse clicks.

⟨*LifeC Class Definition*⟩+≡
```
public:
    ⟨LifeC Mouse Click Declaration⟩
```

The `LifeController` class also contains the member variables which were defined at the beginning of this section of the document.

⟨*LifeC Class Definition*⟩+≡
```
private:
    ⟨LifeC View⟩
    ⟨LifeC Model⟩
```

Once these declarations are all done, we throw all of these together into the class declaration itself. The `LifeController` inherits directly from the `Controller` class of §5.

⟨*LifeC Class Declaration*⟩≡
```
class LifeController : public Controller {
    ⟨LifeC Class Definition⟩
};
```

26.7 The `lifeController.h` file

In this section, we assemble the header file for the `LifeController` class. It is really straightforward since we assembled the class declaration in the previous section. The only thing that we add to the class declaration is that we tuck it into our own name space so that we can keep the global namespace squeaky clean.

⟨*lifeController.h*⟩≡
```
namespace ⟨NameSpace⟩ {
    ⟨LifeC Class Declaration⟩
};
```

26.8 The `lifeController.cpp` file

In this section, we assemble the Life controller source file. It requires the header files for the `Cube` class, the `Controller` class, and the `LifeController` classes.

⟨*lifeController.cpp*⟩+≡
```
#include <SDL.h>
#include "cube.h"
#include "font.h"
#include "soundDev.h"
#include "controller.h"
#include "view.h"
#include "lifeView.h"
#include "life.h"
#include "lifeController.h"
```

After the header files, we include the implementations of the constructor and destructor.

⟨*lifeController.cpp*⟩+≡
```
⟨LifeC Constructor Implementation⟩
⟨LifeC Destructor Implementation⟩
```

After the constructor and destructor, the implementation of the `reset()` method and the game state methods are also included.

⟨*lifeController.cpp*⟩+≡
```
⟨LifeC Reset Implementation⟩
⟨LifeC Game Setting Implementation⟩
```

Then, we include the implementation of the method used to field mouse clicks.

⟨*lifeController.cpp*⟩+≡
⟨*LifeC Mouse Click Implementation*⟩

27 The Life Game View

The namespace inside the Life view class is a concatenation of the general namespace and the name of the Life view class.

⟨*LifeVNameSpace*⟩≡
⟨*NameSpace*⟩`::LifeView`

The Life game view inherits from the generic game view of §6. It displays the current state of the `Life` game.

The Life game stores pointers to the images of the tile pieces to use.

⟨*Life Tiles*⟩≡
```
SDL_Surface* alive;
SDL_Surface* dead;
```

The Life game view also stores a pointer to the font to render some hint text in the sidebar.

⟨*Life Font*⟩≡
```
Font* font;
```

27.1 The Constructor

The constructor for the Life view class takes six arguments. The first is a pointer to the screen, the second is a pointer to the sound device, the third is a pointer to the game cube, the fourth specifies the number of dimensions to employ, the fifth is the skill level, and the sixth is the wrapping mode.

⟨*LifeV Constructor Declaration*⟩≡
```
LifeView(
        SDL_Surface* _screen,
        SoundDev* _sound,
        Cube* _cube,
        unsigned int _dims = 2,
        unsigned int _skillLevel = 0,
        bool _wrap = true
    );
```

The constructor for the Life view class passes all of its arguments to the `View` constructor. Then, it loads the images for alive and dead cells and the font for the hint text.

⟨*LifeV Constructor Implementation*⟩≡

```
⟨LifeVNameSpace⟩::LifeView(
        SDL_Surface* _screen,
        SoundDev* _sound,
        Cube* _cube,
        unsigned int _dims,
        unsigned int _skillLevel,
        bool _wrap
    ) : View( _screen, _sound, _cube, _dims, _skillLevel, _wrap )
{
    this->alive = ::IMG_Load( "../../data/on.png" );
    this->dead = ::IMG_Load( "../../data/off.png" );

    this->font = new Font;
}
```

27.2 The Destructor

The destructor for the life view class simply release the font and images that it loaded above in the constructor.

⟨*LifeV Destructor Declaration*⟩≡

```
~LifeView( void );
```

⟨*LifeV Destructor Implementation*⟩≡

```
⟨LifeVNameSpace⟩::~LifeView( void )
{
    delete font;

    ::SDL_FreeSurface( this->dead );
    ::SDL_FreeSurface( this->alive );
}
```

27.3 The Redraw Methods

The Life view class has a method which allows one to update the entire display area for the game.

⟨*LifeV Redraw Declarations*⟩≡

```
virtual void redraw( void );
```

The redraw function here calls the redraw function on the base class to update the sidebar and the background area of the cube. Then, it runs through each cell in the cube, drawing it. After that, it prints some hints and updates the whole screen.

⟨*LifeV Redraw Implementations*⟩≡

```
void
⟨LifeVNameSpace⟩::redraw( void )
{
    this->View::redraw();

    unsigned int maxIndex
        = ⟨CubeNameSpace⟩::arrayLengths[ this->dims ];

    for ( unsigned int index=0; index < maxIndex; ++index ) {
        this->drawCell( index, false );
    }

    ⟨LifeV Draw Quick Tip Text⟩

    ::SDL_UpdateRect( this->screen, 0, 0, 0, 0 );
}
```

In the sidebar, we're going to scribble some hints for the player on what interactions are available.

⟨*LifeV Draw Quick Tip Text*⟩≡

```
this->font->centerMessage(
        this->screen, false,
        700, 434,
        "Click a cell to toggle it."
    );
this->font->centerMessage(
        this->screen, false,
        700, 474,
        "Right-click or Shift-click"
    );
this->font->centerMessage(
        this->screen, false,
        700, 498,
        "anywhere to pass time."
    );
```

The Life view class has a method which allows one to update a single cell of the cube by index.

⟨*LifeV Redraw Declarations*⟩+≡

```
virtual void redraw( unsigned int index );
```

This method simply uses the method defined next to draw the single cell in question.

⟨*LifeV Redraw Implementations*⟩+≡

```
void
⟨LifeVNameSpace⟩::redraw( unsigned int index )
{
    this->drawCell( index );
}
```

The Life view class has a method to draw a single cell of the cube. It uses this method in each of the above methods.

⟨*LifeV Private Draw Declaration*⟩≡

```
void drawCell( unsigned int index, bool update = true );
```

To draw a single cell, this method retrieves the screen coordinates of the cell from the conversion method in the base class. Then, it prepares a rectangle to fill for the cell. Then, depending on the state of the cell in the game cube, it either draws the region off or on.

⟨*LifeV Private Draw Implementation*⟩≡

```
void
⟨LifeVNameSpace⟩::drawCell(
        unsigned int index, bool update
    )
{
    unsigned int xx;
    unsigned int yy;

    View::cellToScreen( index, this->dims, &xx, &yy );

    ⟨LifeV Prepare Single Cell Rect⟩

    if ( (*this->cube)[ index ] == 0 ) {
        ::SDL_BlitSurface( this->dead, 0, this->screen, &rect );
    } else {
        ::SDL_BlitSurface( this->alive, 0, this->screen, &rect );
    }
    if ( update ) {
        ::SDL_UpdateRect( this->screen, xx, yy, SQUARE, SQUARE );
    }
}
```

The rectangle that will be filled to represent the cell simply starts at the starting coordinates of the rectangle and goes almost the full size of the cell. It doesn't go quite to the edge so that one can clearly see the break between cells.

⟨*LifeV Prepare Single Cell Rect*⟩≡
```
SDL_Rect rect;
rect.x = xx;
rect.y = yy;
rect.w = SQUARE;
rect.h = SQUARE;
```

27.4 The `LifeView` class

In this section, we assemble the `LifeView` class from the pieces in the sections above.

We include, in the `LifeView` class, the constructor, the destructor and the redraw methods.

⟨*LifeV Class Definition*⟩≡
```
public:
    ⟨LifeV Constructor Declaration⟩
    ⟨LifeV Destructor Declaration⟩
    ⟨LifeV Redraw Declarations⟩
private:
    ⟨LifeV Private Draw Declaration⟩
```

We include the variables that are used in the life view class.

⟨*LifeV Class Definition*⟩+≡
```
private:
    ⟨Life Tiles⟩
    ⟨Life Font⟩
```

Once these declarations are all done, we throw all of these together into the class declaration itself. The `LifeView` inherits directly from the `View` class of §6.

⟨*LifeV Class Declaration*⟩≡
```
class LifeView : public View {
    ⟨LifeV Class Definition⟩
};
```

27.5 The `lifeView.h` file

In this section, we assemble the header file for the `LifeView` class. It is really straightforward since we assembled the class declaration in the previous section. The only thing that we add to the class declaration is that we tuck it into our own name space so that we can keep the global namespace squeaky clean.

⟨*lifeView.h*⟩≡

```
namespace ⟨NameSpace⟩ {
    ⟨LifeV Class Declaration⟩
};
```

27.6 The `lifeView.cpp` file

In this section, we assemble the Life view source file. It requires the `SDL` headers for dealing with surfaces, the screen, blitting, and loading images. It requires the header files for the `Cube` class, the `Font` class, the `SoundDev` class, the `View` class, and the `LifeView` class itself.

⟨*lifeView.cpp*⟩≡

```
#include <SDL.h>
#include <SDL_image.h>
#include "cube.h"
#include "font.h"
#include "soundDev.h"
#include "view.h"
#include "lifeView.h"
```

After the header files, we include the implementations of the constructor, the destructor, and the redraw methods.

⟨*lifeView.cpp*⟩+≡

```
⟨LifeV Constructor Implementation⟩
⟨LifeV Destructor Implementation⟩
⟨LifeV Redraw Implementations⟩
⟨LifeV Private Draw Implementation⟩
```

Part IX
The Main Program

28 The main loop of the program

Note, this uses `stdio` instead of the C++ `iostream` because using the `iostream` functions really causes a jump in the size of the program. I know that `iostream` is the "right" thing to do, but the overhead in program size isn't worth it to me for this contest.

28.1 Initializing the SDL Library

We have to initialize all of the SDL components that we are going to use. In our case, it's just the video and the audio.

⟨*Main Initialize SDL*⟩≡
```
unsigned int initFlags = 0;
initFlags |= SDL_INIT_VIDEO;
initFlags |= SDL_INIT_AUDIO;
```

After we prepare the flags, we simply call the init routine for the SDL library.

⟨*Main Initialize SDL*⟩+≡
```
if ( ::SDL_Init( initFlags ) < 0 ) {
    fprintf( stdout, "Failed to init SDL: %s\n", ::SDL_GetError() );
    return __LINE__;
}
```

If we succeeded in initializing SDL, then we'll have to uninitialize it when we exit.

⟨*Main Quit SDL*⟩≡
```
::SDL_Quit();
```

28.2 Initializing the SDL Video Mode

The following code initializes the SDL video mode. We require that the screen be 800x600. All of the code in the `View` class assumes that, and all of the images are sized for those dimensions. At the moment, we're forcing a 24-bit depth. Maybe we should use the `SDL_ANYFORMAT` flag instead. But, for the moment, we'll go with this.

⟨*Main Initialize Video Mode*⟩≡

```
unsigned int videoFlags = 0;

videoFlags |= SDL_SWSURFACE;
if ( argc <= 1 ) {
    videoFlags |= SDL_FULLSCREEN;
}

SDL_Surface* screen = ::SDL_SetVideoMode(
        800, 600, 24, videoFlags
    );

if ( screen == 0 ) {
    fprintf( stdout, "Failed to set video mode: %s\n",
            ::SDL_GetError()
        );
    return __LINE__;
}
```

Once we've initialized the screen, we set up the text for the title bar of the window and the icon.

⟨*Main Initialize Video Mode*⟩+≡

```
::SDL_WM_SetCaption( "54321 v1.0.2001.11.16", "54321" );
```

After the screen has been initialized, we set the gamma to brighten things up a bit.

⟨*Main Initialize Video Mode*⟩+≡

```
::SDL_SetGamma( 1.6, 1.6, 1.6 );
```

28.3 Initializing the SDL Audio Device

Here, we try to open the sound device. If this fails, then we just forget about sound altogether.

⟨*Main Initialize Audio Device*⟩≡

```
⟨NameSpace⟩::SoundDev* soundDev = new ⟨NameSpace⟩::SoundDev;

if ( ! soundDev->isOpened() ) {
    delete soundDev;
    soundDev = 0;
}
```

28.4 The Seeding the Random Numbers

To ensure that we start with different games each time, we will initialize the random number generator based upon the process id and the number of ticks that have happened up until this point.

⟨*Main Seed Random Number Generator*⟩≡

```
::srandom( getpid() ^ ::SDL_GetTicks() );
```

28.5 The Event Loop

The main loop passes mouse events into the current controller. It catches user events to switch modes. And, it catches keyboard events to do screen captures unless we're in release mode.

⟨*Main Event Loop*⟩≡

```
bool done = false;

SDL_Event event;

while ( ! done && ::SDL_WaitEvent( &event ) ) {
    switch ( event.type ) {
    case SDL_MOUSEBUTTONDOWN:
    case SDL_MOUSEBUTTONUP:
        ⟨Main Handle Mouse Click⟩
        break;
    case SDL_USEREVENT:
        delete controller;
        switch ( event.user.code ) {
            ⟨Main Handle State Change⟩
        }
        break;
#ifndef NDEBUG
    case SDL_KEYUP:
        ⟨Main Handle Key Up⟩
        break;
#endif
    case SDL_QUIT:
        done = true;
        break;
    }
}
```

The salient portions of the mouse click event are pulled out of the event structure and passed into the current controller's mouse click handler. For those without a two-button mouse, a meta key or a shift key will simulate a higher button number.

⟨*Main Handle Mouse Click*⟩≡

```
if ( controller != 0 ) {
    bool isMouseUp = ( event.type == SDL_MOUSEBUTTONUP );
    unsigned int xx = event.button.x;
    unsigned int yy = event.button.y;
    unsigned int buttonNumber = event.button.button;
    unsigned int mask = KMOD_META | KMOD_SHIFT;

    if ( ( ::SDL_GetModState() & mask ) != 0 ) {
        ++buttonNumber;
    }

    controller->handleMouseClick(
            isMouseUp, xx, yy, buttonNumber
        );
}
```

The main menu generates events which tell the main loop to load start a new controller. These events are simple user events. The event's code tells which game to pick.

⟨*Main Handle State Change*⟩+≡

```
case ⟨MainMenuVNameSpace⟩::FLIPFLOP:
    controller = new ⟨NameSpace⟩::FlipFlopController(
                 screen, soundDev, &cube
             );
    break;
case ⟨MainMenuVNameSpace⟩::BOMBSQUAD:
    controller = new ⟨NameSpace⟩::BombSquadController(
                 screen, soundDev, &cube
             );
    break;
case ⟨MainMenuVNameSpace⟩::MAZERUNNER:
    controller = new ⟨NameSpace⟩::MazeController(
                 screen, soundDev, &cube
             );
    break;
case ⟨MainMenuVNameSpace⟩::PEGJUMPER:
    controller = new ⟨NameSpace⟩::PegController(
                 screen, soundDev, &cube
             );
    break;
case ⟨MainMenuVNameSpace⟩::TILESLIDER:
    controller = new ⟨NameSpace⟩::TileController(
                 screen, soundDev, &cube
             );
    break;
default:
    controller = new ⟨NameSpace⟩::MainMenuController(
                 screen, &cube
             );
    break;
```

If we're not compiling for release mode, then the comma key causes a screen grab into a BMP.

⟨*Main Handle Key Up*⟩≡

```
if ( event.key.keysym.sym == SDLK_COMMA ) {
    SDL_SaveBMP( screen, "../../screengrab.bmp" );
}
```

28.6 The `main.cpp` file

This next chunk defines the main routine of the program. It incorporates all of the previous portions of this section.

⟨*Main SDL-main*⟩≡

```
extern "C" int
SDL_main( int argc, char** argv )
{
    ⟨Main Initialize SDL⟩
    ⟨Main Initialize Video Mode⟩
    ⟨Main Initialize Audio Device⟩

    ⟨Main Seed Random Number Generator⟩

    ⟨NameSpace⟩::Cube cube;
    ⟨NameSpace⟩::Controller* controller = 0;

    SDL_Event change;
    change.type = SDL_USEREVENT;
    change.user.code = -1;
    ::SDL_PushEvent( &change );

    ⟨Main Event Loop⟩

    delete controller;
    delete soundDev;

    ⟨Main Quit SDL⟩
    return 0;
}
```

The main routine requires the `<stdio.h>` header for the `fprintf()` calls it makes. It needs `<stdlib.h>` for the declaration of `srandom()`. It needs `<unistd.h>` for the declaration of `getpid()`. It requires the `<SDL.h>` header for all of the Simple DirectMedia Layer calls it invokes. It requires the `"cube.h"` for the definition of the `Cube` class. It requires the `"font.h"` as a prerequisite for some of the `View` subclasses. It requires the `"soundDev.h"` for the interface to the SDL audio device. And, it requires the `"controller.h"` for the interface to the game controllers. It requires the `"view.h"` file so that it can include the game-specific controllers which use the game-specific `View` derivatives.

⟨*main.cpp*⟩≡

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <SDL.h>
#include <SDL_keysym.h>
#include "cube.h"
#include "font.h"
#include "soundDev.h"
#include "controller.h"
#include "view.h"
```

After that, it includes the classes used for the main menu.

⟨*main.cpp*⟩+≡

```
#include "mainmenuView.h"
#include "mainmenuController.h"
```

After that, it includes the classes used for the flip flop game.

⟨*main.cpp*⟩+≡

```
#include "flipflopView.h"
#include "flipflop.h"
#include "flipflopController.h"
```

After that, it includes the classes used for the bomb squad game.

⟨*main.cpp*⟩+≡

```
#include "bombView.h"
#include "bomb.h"
#include "bombController.h"
```

After that, it includes the classes used for the maze runner game.

⟨*main.cpp*⟩+≡

```
#include "mazeView.h"
#include "maze.h"
#include "mazeController.h"
```

After that, it includes the classes used for the peg jumper game.

⟨*main.cpp*⟩+≡

```
#include "pegView.h"
#include "peg.h"
#include "pegController.h"
```

After that, it includes the classes used for the tile slider game.

⟨*main.cpp*⟩+≡
```
#include "tileView.h"
#include "tile.h"
#include "tileController.h"
```

After that, it includes the classes used for the life game.

⟨*main.cpp*⟩+≡
```
#include "lifeView.h"
#include "life.h"
#include "lifeController.h"
```

The routine from the previous subsection is the only one included in the source file `main.cpp`.

⟨*main.cpp*⟩+≡
⟨*Main SDL-main*⟩

Part X
Architecture-Specific Code

29 Darwin-specific code

This code is entirely snagged from code generated by the first SDL port to Mac OS X by Darrell Walisser (dwaliss1@purdue.edu).

29.1 The Objective C application object

This portion declares the objective C object for the SDL program.

⟨*Darwin SDLMain Interface*⟩≡
```
@interface SDLMain : NSObject
{
}
- (IBAction)quit:(id)sender;
@end
```

This portion catches the quit event sent by Cocoa to the application. It turns it into an SDL_QUIT event.

⟨*Darwin SDLMain Quit*⟩≡
```
- (void) quit:(id)sender
{
    SDL_Event event;
    event.type = SDL_QUIT;
    SDL_PushEvent( &event );
}
```

This portion gets the program running in the right directory so that it has all of its resources available to it. We don't use many of those resources, but we need the ones that are there.

⟨*Darwin SDLMain Setup Dir*⟩≡

```
- (void) setupWorkingDirectory
{
    char parentDir[ MAXPATHLEN + 1 ];
    char *ch;

    strncpy( parentDir, gArgv[ 0 ], MAXPATHLEN );
    ch = parentDir;

    while ( *ch != '\0' ) {
        ++ch;
    }

    while ( *ch != '/' ) {
        --ch;
    }

    *ch = '\0';

    assert( chdir( parentDir ) == 0 );
    assert( chdir( "../../../" ) == 0 );
}
```

This method gets called when the application is done launching. This calls the `setupWorkingDirectory` method defined above and then calls the `SDL_main` program.

⟨*Darwin SDLMain Finish*⟩≡

```
- (void) applicationDidFinishLaunching:(NSNotification*)note
{
    int status;
    [ self setupWorkingDirectory ];

    status = SDL_main( gArgc, gArgv );
    exit( status );
}
```

29.2 The `ObjectiveMain` routine

This next section is the main routine for the Objective C portion of the code. It simply gets the command-line arguments ready to be sent to `SDL_main()`. Then, it calls the application main.

⟨*Darwin SDLMain ObjectiveMain*⟩≡

```
int
ObjectiveMain( int argc, char** argv )
{
    int ii;

    if ( argc >= 2 && strncmp( argv[1], "-psn", 4 ) == 0 ) {
        gArgc = 1;
    } else {
        gArgc = argc;
    }

    gArgv = (char**)malloc( sizeof(*gArgv) * (gArgc+1) );
    assert( gArgv != NULL );
    for ( ii=0; ii < gArgc; ++ii ) {
        gArgv[ ii ] = argv[ ii ];
    }

    gArgv[ ii ] = NULL;

    NSApplicationMain( argc, argv );

    return 0;
}
```

29.3 The `Darwin-main.m` file

The above bits are all assembled to form an Objective-C source file. The source file includes the SDL header, some Cocoa headers, and a few headers that it needs to do the directory changes above.

⟨*Darwin-main.m*⟩≡

```
#import "SDL.h"
#import <Cocoa/Cocoa.h>
#import <sys/param.h>
#import <unistd.h>
#import <stdlib.h>
```

After that, the source file includes the definition of the SDL application object interface.

⟨*Darwin-main.m*⟩+≡

⟨*Darwin SDLMain Interface*⟩

The source file then defines some global variables used to track the command-line arguments.

⟨*Darwin-main.m*⟩+≡
```
static int gArgc;
static char **gArgv;
```

After that, the source file includes the implementation of the SDL application object defined above.

⟨*Darwin-main.m*⟩+≡
```
@implementation SDLMain
⟨Darwin SDLMain Quit⟩
⟨Darwin SDLMain Setup Dir⟩
⟨Darwin SDLMain Finish⟩
@end
```

And, the source file includes the `ObjectiveMain()` routine from §29.2.

⟨*Darwin-main.m*⟩+≡
```
⟨Darwin SDLMain ObjectiveMain⟩
```

29.4 The `Darwin-main-help.cpp` file

To get things started, we actually define main in a C++ file. If we don't do this, then global objects don't end up getting constructed. The main routine here simply turns around and invokes the `ObjectiveMain()` from §29.2

⟨*Darwin-main-help.cpp*⟩≡
```
#ifdef main
    #undef main
#endif

extern "C" int ObjectiveMain( int, char*[] );

int
main( int argc, char* argv[] )
{
    return ObjectiveMain( argc, argv );
}
```