14 Introduction to C++ and OpenGL

14.1 Some C++ Basics

Here are a few tips for programming in C++, assuming you're already familiar with Java. C++ and Java have similar syntax, but there are several significant differences in how they work.

14.1.1 Types of variables

Unlike languages such as Java, C++ gives the programmer direct access to memory used by variables. Not only are the *contents* of a variable important, but *where* that variable is stored. As a result, there are a number of ways variables can be treated.

Value variables store their data directly in their memory location.

```
int a;
a = 1;    // Now a is set to 1
```

Pointer variables (declared by prepending a variable name with an asterisk) store the address of another memory location. A variable name is prepended with an ampersand to get its address, and a pointer is dereferenced to the memory location it points to by prepending it with an asterisk. Unlike a reference variable, a pointer variable can be null, which means it does not point to a valid memory location.

```
int *c = &a;
*c = 3;    // Now a == 3 is true
```

Reference variables (declared by prepending a variable name with an ampersand) store their data in the memory location used by another variable. These must be initialized during declaration to an existing variable and cannot be changed to refer to a different variable later. Once declared, a reference variable can be used like a regular value variable, only changing it also changes the value variable it references.

```
int &b = a;
b = 2;    // Now a == 2 is true
```


Reference variables are not used very often in C++ programming. However, reference is very common in C++, where you define an argument to a function as a reference variable. The following function changes the value of its argument, whereas it would not if there:

```
void function(int & b)
{
 b = 2;
}
```

For local or member variables, there are a few things to keep in mind when choosing the type of the variable. Value variables are most often used for small objects, since the entire object will be stored on the stack, which has a limited capacity. For example, initializing a huge array as a value variable should be avoided. However, since no new memory needs to be allocated for a value variable, they are very quick to initialize. Pointers are typically used for larger objects and allocated in memory using the `new` operator. Since C++ performs no automatic memory management, objects allocated with `new` must be freed with `delete` when they are no longer needed. Furthermore, a pointer must be used if an object is going to be treated polymorphically. Reference variables are rarely used as locals or members for a class.

For function parameters, value variables are used for smaller objects whose value should not change, since a copy of the object is made. When you want to avoid the overhead of copying an object or allow the contents of a variable to be changed from within a function, a reference variable should be used as a parameter. Pointer variables also allow a variable outside a function's scope to be modified, but the programmer must be careful with the dereferencing semantics when using pointers. In particular, a pointer could be null, so the program must check for this case and treat it appropriately before attempting to dereference the pointer.

14.1.2 Object-oriented programming

Objects (classes) in C++ have some important differences from Java classes.

Inheritance in Java only allows each class to have a single parent class (or super class), such as

```
class ChildClass extends ParentClass
```

C++ supports multiple inheritance, using the following syntax:

```
class ChildClass : public ParentClass1, public ParentClass2, ...
```


To avoid multiple instances of a single base class, the common base class must be declared as `virtual`.

Object members are accessed in Java using the dot operator. In C++, the dot operator only works for value and reference variables. For pointers, the arrow operator is used, as follows:

```
SomeClass *someObject = new SomeClass();
someObject->someMember();    // Dereference and access a member
(*someObject).someMember();  // An equivalent statement
```

Abstract methods are created in Java by using the `abstract` keyword. Abstract base classes in C++ are created by declaring at least one member function as *pure virtual*, achieved by using the virtual keyword and ending the function definition with `= 0`.

Interfaces are implemented in Java by using the `interface` keyword when defining a class. C++ does not directly support interfaces, but abstract base classes can be created with all pure virtual functions.

Parent class constructors are automatically called in Java if `super()` is not the first line of a constructor body. In C++, if no call is made to a parent class constructor, an implicit call is made to the default constructor. To explicitly call a parent's constructor, use

```
this->ParentClass::ParentClass().
```

Polymorphism in Java is accomplished with inheritance and overriding of non-`final` methods. C++ requires that a function specifically be declared `virtual` to override.

Libraries can be used in Java by using the `import` keyword. C++ relies on the `#include` preprocessor directive, which treats all of the text in the included header file as though it were part of the including source file. Headers usually contain class and function declarations without definitions of their bodies. When compiling, a linking step is used to include the appropriate machine code for the library implementations in the final program.

The program entry point in Java is the `Main` method, which can be located in any class. C++ expects this function to exist outside any classes, with the following signature:

```
int main(int argc, char** argv)
```


Other constructors within a class are called in Java with `this(...)`. C++ does not have a mechanism for such a shortcut. Instead, the constructor logic can be placed in a private function called by all of the constructors.

Generics in Java simplify development by compiling undetermined types to be `Objects` and performing casting automatically at runtime when user code gives the specific types of a generic class. C++ uses templates, which are resolved at compile time instead. Templates can be used on classes with this syntax:

```
template <class T> class ClassName
```

Similarly, templates can be used for functions with this syntax:

```
template <class T> ReturnType FunctionName(T argName, ...)
```

14.2 Getting Started with OpenGL

Three libraries are used for the programming assignments throughout this course: OpenGL, GLU, and GLUT. All three of these are portable to many platforms, including Linux and Windows.

The *OpenGL Programmer's Guide* and the *OpenGL Reference Manual* are absolutely essential books for OpenGL programming. If you have questions about individual functions or how to use them, look in these books. They are available online or to purchase.

OpenGL provides a consistent interface to the underlying graphics hardware. This abstraction allows a single program to run a different graphics hardware easily. A program written with OpenGL can even be run in software (slowly) on machines with no graphics acceleration. OpenGL function names always begin with `gl`, such as `glClear()`, and they may end with characters that indicate the types of the parameters, for example `glColor3f(GLfloat red, GLfloat green, GLfloat blue)` takes three floating-point color parameters and `glColor4dv(const GLdouble *v)` takes a pointer to an array that contains 4 double-precision floating-point values. OpenGL constants begin with `GL`, such as `GL_DEPTH`. OpenGL also uses special names for types that are passed to its functions, such as `GLfloat` or `GLint`, the corresponding C types are compatible, that is `float` and `int` respectively.

GLU is the OpenGL utility library. It contains useful functions at a higher level than those provided by OpenGL, for example, to draw complex shapes or set up cameras. All GLU functions are written

on top of OpenGL. Like OpenGL, GLU function names begin with `glu`, and constants begin with `GLU`.

GLUT, the OpenGL Utility Toolkit, provides a system for setting up callbacks for interacting with the user and functions for dealing with the windowing system. This abstraction allows a program to run on different operating systems with only a recompile. Glut follows the convention of prepending function names with `glut` and constants with `GLUT`.

14.2.1 Writing an OpenGL Program with GLUT

An OpenGL program using the three libraries listed above must include the appropriate headers. This requires the following three lines:

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
```

Before OpenGL rendering calls can be made, some initialization has to be done. With GLUT, this consists of initializing the GLUT library, intializing the display mode, creating the window, and setting up callback functions. The following lines initialize a full color, double buffered display:

```
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
```

Double buffering means that there are two buffers, a front buffer and a back buffer. The front buffer is displayed to the user, while the back buffer is used for rendering operations. This prevents flickering that would occur if we rendered directly to the front buffer.

Next, a window is created with GLUT that will contain the viewport which displays the OpenGL front buffer with the following three lines:

```
glutInitWindowPosition(px, py);
glutInitWindowSize(sx, sy);
glutCreateWindow(name);
```

To register callback functions, we simply pass the name of the function that handles the event to the appropriate GLUT function.

```
glutReshapeFunc(reshape);
glutDisplayFunc(display);
```


Here, the functions should have the following prototypes:

```
void reshape(int width, int height);
void display();
```

In this example, when the user resizes the window, `reshape` is called by GLUT, and when the display needs to be refreshed, the `display` function is called. For animation, an idle event handler that takes no arguments can be created to call the display function to constantly redraw the scene with `glutIdleFunc`. Once all the callbacks have been set up, a call to `glutMainLoop` allows the program to run.

In the display function, typically the image buffer is cleared, primitives are rendered to it, and the results are presented to the user. The following line clears the image buffer, setting each pixel color to the clear color, which can be configured to be any color:

```
glClear(GL_COLOR_BUFFER_BIT);
```

The next line sets the current rendering color to blue. OpenGL behaves like a state machine, so certain state such as the rendering color is saved by OpenGL and used automatically later as it is needed.

```
glColor3f(0.0f, 0.0f, 1.0f);
```

To render a primitive, such as a point, line, or polygon, OpenGL requires that a call to `glBegin` is made to specify the type of primitive being rendered.

```
glBegin(GL_LINES);
```

Only a subset of OpenGL commands are avaiable after a call to `glBegin`. The main command that is used is `glVertex`, which specifies a vertex position. In `GL_LINES` mode, each pair of vertices define endpoints of a line segment. In this case, a line would be drawn from the point at (`x0`, `y0`) to (`x1`, `y1`).

```
glVertex2f(x0, y0);
glVertex2f(x1, y1);
```

A call to `glEnd` completes rendering of the current primitive.

```
glEnd();
```

Finally, the back buffer needs to be swapped to the front buffer that the user will see, which GLUT can handle for us:

```
glutSwapBuffers();
```

