Object Oriented Mobile Game Programming with Rude Engine

Robert Rose

March 14, 2004

Abstract

This paper covers the essential concepts of Object Oriented Programming (OOP) in the context of game programming, and introduced the *Rude Engine* 3D game programming framework for mobile platforms.

1 Introduction

Object Oriented Programming (OOP) is often referred to as a "new paradigm" for developing software, when in fact OOP is really isn't that much of a shift in thinking at all. Thinking in terms of objects is probably the most natural way we as humans can think can think about programming. OOP is also extends itself very naturally to game development, as most games have very clear object-like elements in game play that can directly correlated with objects in code. This paper explains several core concepts of OOP, introduces several issues with using OOP on mobile platforms, and outlines an OOP 3D game programming framework for mobile platforms, *Rude Engine*.

2 Thinking Object Oriented

The world we live in is very complex. As humans, our brain helps us deal with complexity by obscuring details from us and organizing things into groups so we can more quickly analyze and adapt to the world around us. For example, in the act of driving, we expect our brains to help us get us to our destination

in a safe efficient manner. When we see another vehicle approaching and we try to avoid it, our brain does not tell us "there is a 4000 lb mass constructed of steel, rubber and plastics arranged in a 4-wheel configuration powered by a 6-cylinder, 4-cycle gasoline engine controlled by an accelerator petal, brake pedal, transmission... and it's trajectory will intercept our own vehicle in 2.3 seconds if we don't rotate our steering wheel clockwise to avoid it!" Instead, our brain simply tells us there is a *car* approaching, and we need to *get out of the way*. If our brain's alerted us to all of the incredible details of the world we live in constantly, we would surely not be able to cope with it.

Object Oriented Programming's goals are much the same as our brain's: obscure details and better organize things so we can think about them more easily. Programming is a complicated activity which can involve many thousands of lines of code and hundreds of specific algorithms. It is rarely expected of anyone to keep a large program "in their head" at any time, or to understand how specific portions of a large program work. Object Oriented Programming can significantly reduce the complexity involved in understanding a program by simplifying large portions of a program.

2.1 Objects

OOP simplifies programming by forcing us to think in terms of interacting *objects*. An object can be thought of as a small, simple computer program. The object exposes to us a small set of actions that it can perform, and more importantly, we do not care how it performs these actions. To build a new computer program we assemble different objects together and let them interact. By organizing our program as objects we can more easily understand our program and explain it to others.

Thinking of a computer program as a set of interacting objects is best explained by an example. In a restaurant there are customers, servers, chefs, and managers. When the customer enters the restaurant, they request a table from the manager. Later a server comes to take the order from the customer, and the order is given to the chef for preparation. When the order is prepared, the meal is delivered by the server to the customer. When the customer finishes the meal the customer pays either the server or the manager depending upon the establishment.

Each of these "actors" in the restaurant (customers, servers, etc.), can all be thought of as objects. (The *order* could also be thought of as an object). The customer interacts with a manager and a server, the server with the chef,

and so-on. When the customer places an order with the server, the customer doesn't need to know how the order is realized as a meal[1], the server takes care of it for the customer. One possible view of the relationships between these objects in the restaurant world is given in Figure 1.

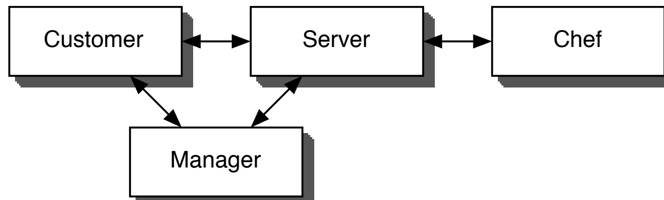

Figure 1: Objects in the Restaurant World

Drawing analogies between the world we live in and problems in the OOP space is useful because it helps us better to think in an object oriented manner. As humans, we essentially think of things in terms of objects, and because of this, it is easier for us to think of programming as being object oriented. If you don't quite grasp this concept yet, give it time, transitioning to thinking in objects from traditional programming doesn't happen overnight.

2.2 Classes and Methods

In OOP, we create objects by writing *classes*. A class is the definition of how an object will behave. When the program runs, we create new objects by saying, "I want a new object, and this class defines it's behavior." While the program is running objects always keep track of what type of class they are.

Classes can be thought of as having two parts, their *interface* and their *implementation*. The interface is what other objects care about; the interface describes how the class is expected to interact with other objects. The implementation describes the actual behavior of the class, this is the code that runs the class.

So that other objects may interact with a class, classes define *methods*. Methods can be thought of as "class functions." A method is a special type of function that is tied to a particular class. When other objects interact with an object, they do so by invoking the methods of the object. The code for these methods are stored inside the object's class.

[1]In some establishments this may be considered a good thing.

The difference between classes and objects can be a little tricky to grasp at first. Let's look at the restaurant example mentioned previously. In the restaurant the "server" object has the responsibility of explaining the menu to the customer, taking the customer's order, and delivering the customer's meal from the chef. These three main tasks can be thought of as the server's interface. How the server actually performs these tasks can be thought of as the server's implementation. Now if we were to write a program that ran the restaurant, we would describe the server's behavior in a class. When the program runs, we create one or more *instances* of the server class–these are the server objects.

Following this example further we see that the three main tasks of the server can be thought of as the methods of the server. When a customer wants to place an order, the customer object would invoke the "take order" method of their server object. The server object handles the method invocation by using it's server class to execute the code necessary to take the order.

2.3 Inheritance

A powerful notion in OOP is the idea of *inheritance*. When one class is said to "inherit" from another it means that the class implements a more specific interface of the other. The class that is being inherited is called the *parent* and the class that is inheriting is the *child*. Inheritance is an extremely useful tool in OOP and can be used in many different ways. Budd [1] defines many ways inheritance can be used, four of which are described below:

1. *Specification*. The parent has little or no implementation, it only serves to provide an interface that the child is expected to implement. From the restaurant example, there may be many different types of server objects (fast servers, slow servers, poor servers, etc.), but all servers are expected to implement the same interface. If we created these other server types by inheriting from a parent server type, we would be using inheritance for specification, because the parent would describe how the children would be expected to behave.

2. *Extension*. The child does not redefine the behavior of the parent, but adds new behaviors.

3. *Combination*. An inheritance technique available in some OOP languages is multiple-inheritance, where a child can inherit from multiple

parents.

4. *Construction.* The child inherits from the parent only so it can borrow code from the parent. Inheritance for construction is generally considered bad practice. If one class needs to borrow code from another it should declare an instance of the other class as an object within itself rather than use inheritance.

Child classes may combine any of these styles of inheritance at any time, there are no rules for how inheritance needs to be used.

2.4 Polymorphism

The term *polymorphism* is used in OOP to mean many different things, but generally refers to some aspect of an object having several meanings. An object's method could be "polymorphic," or you could have an object that is declared as a "polymorphic variable." Below are three ways Budd [1] explains polymorphism can be used in OOP[2]:

1. *Overloading.* An "overloaded" method is one that has two or more different versions. You may have a method "foo" for example, and one version takes one parameter and another that takes too; this would mean "foo" is overloaded.

2. *Overriding.* Earlier we discussed inheritance. If a child inherits from a parent and redefines one of the parent's methods, then it said to have "overridden" the method.

3. *Polymorphic variable.* A polymorphic variable is a variable that is declared to be of one type but actually is of another type. If a child inherits from a parent, then we can declare a new instance of the child but store it as an instance of the parent, and only we are the wiser.

3 Mobile Game Development

Mobile platforms are unique in that, unlike traditional computer systems we have grown accustomed to developing for recently, mobile platforms are

[2]Budd actually gives a fourth way known as *generics*, but most mobile platforms do not support this style of polymorphism.

tremendously constrained. Mobile platforms carry with them a large set of limitations, among them:

1. *Speed*. Mobile platforms are slower than regular computer systems. Efficiency, an art long since forgotten for most game development, has been reinvigorated on mobile platforms.

2. *Memory*. Mobile platforms generally have significantly less memory than regular computers. Memory constraints can appear in unusual situations on mobile devices.

3. *Tool-chain*. Regular development tools rarely are carried over onto mobile platforms. Developing for mobile platforms means you are going to need to learn a lot of new and different development processes.

Because of these and other limitations, developing object oriented applications on mobile platforms is significantly different. Mainstream object oriented environments such as Java and .NET have undergone severe modifications to be runnable on mobile platforms in the form of "Java 2 Micro Edition," and ".NET Compact Framework," etc. While these environments largely enhance portability across mobile platforms, they do no offer the speed necessary for 3D game development so are not covered by this paper. Instead, this paper only address object oriented development with C++.

3.1 Windows Mobile

Today's Windows Mobile suite of mobile platforms come in a variety of flavors: Pocket PC, Pocket PC 2002, Pocket PC 2003, SmartPhone 2002, and SmartPhone 2003.

The 2002 (and previous) varieties represent the largest market segment, however they have severely crippled C++ implementations. The largest limitation of which is the lack of C++ template support, so there is no STL implementation available.

3.2 Palm OS

Palm OS has no standard C library.

3.3 Symbian

Symbian has very interesting operating system extensions to accelerate C++ [2]. However, to remain platform independent, they are best avoided. If your application is Symbian-only, then exploit them as much as possible, as they yield much better performance.

The Symbian compiler does not pay attention to byte alignment issues, they leave it up to you to take care of. Be very careful how you access memory on Symbian, be sure to read the documentation on multi-byte alignment.

3.4 Best Practices

1. *No objects on stack.* Avoid declaring objects on the stack. Symbian has a finicky memory management system and will screw up C++ objects you declare on the stack. Use the `new` and `delete` operators instead.

2. *No globals.* You've heard this before as a general programming suggestion, but on Symbian it's a requirement. Symbian does not allow variable to be declared in your programs data segment. Use singleton classes instead.

3. *Don't use STL.* Although Windows Mobile 2003 has STL support, the rest of Windows Mobile does not. Stay away from STL if you want to remain platform independent.

4. *Use a cross-platform framework.* Write generic code that talks completely through a platform independent, generic framework. Rude Engine, discussed in the next section, is one such framework!

4 Rude Engine

Creating a 3D game that is portable across mobile platforms is a daunting task. Reaching this goal has tremendous benefits however–portability across multiple mobile platforms means your game will reach a wider audience and hopefully sell more copies. *Instant portability*, being able to quickly compile your game for multiple platforms simultaneously, has even more benefits–higher quality code due to the diversity of compilers, and easier bug finding due to the diversity of memory models on mobile platforms. The goal of

Rude Engine is for your game to be instantly portable, and reap all of the benefits portability across mobile platforms has to offer.

Rude Engine is a *framework* for game-oriented applications. Rude Engine provides an application and graphics framework for games, along with platform-independent abstractions for graphics primitives and sound. By using the platform-independent abstractions that Rude Engine offers your game will be instantly portable across Windows Desktop, Windows Mobile SmartPhone, Windows Mobile Pocket PC, Palm OS, and Symbian.

Rude Engine provides a tremendous amount of foundation for you to build great games on top of. Developing an application using Rude Engine is as simple as deriving from the base class `RudeGame` and implementing just a few of the base methods so your game can draw output and handle user input. Rude Engine includes many helper classes to help you accomplish this! These classes and their usage is described in the following sections.

4.1 RudeGame

```
virtual void Render(RudeSurface &screen, float delta, float aspect);
virtual void KeyDown(RudeKey k);
virtual void KeyUp(RudeKey k);
virtual void StylusDown(RudeScreenVertex &p);
virtual void StylusMove(RudeScreenVertex &p);
virtual void StylusUp(RudeScreenVertex &p);
virtual void Pause();
virtual bool Done();
static  float GetTime();
```

The heart of Rude Engine is the `RudeGame` class, the base class from which all game's that wish to use Rude Engine must derive from. When you write your own `RudeGame` child class, you may implement any number of `RudeGame`'s virtual methods.

The parameters of all of `RudeGame`'s methods are designed to insure platform independence. For example, the `RudeSurface` class (discussed later), is a polymorphic variable that holds the surface object being used for the current platform. `RudeKey` is a generic data type that holds a key identifier. `RudeScreenVertex` is also a generic data type that holds a mouse (or stylus) point on the screen. As long as your game sticks to use these generic types you will have a portable game!

4.2 RudeSurface

```
static RudeSurface * CreateSurface(RudeSurface *previous);
static RudeSurface * CreateSurface(long width,
long height, RudeSurface *previous);

long GetWidth();
long GetHeight();
RudeRect& GetRect();

virtual void FillRect(RudeRect *rect, RudeColor *color);
virtual void FillRect(RudeRect *rect, RudeColor *color, int alpha);
virtual void Clear();
virtual void DrawLine(long x1, long y1, long x2, long y2, RudeColor *color);
virtual void DrawLineAA(long x1, long y1, long x2, long y2, RudeColor *color);
virtual void DrawChars(long x, long y, TCHAR *text, RudeColor *color);
virtual void SetPixel(long x, long y, RudeColor *color, int alpha);
virtual void SetPixel(long x, long y, RudeColor *color);
virtual void GetPixel(long x, long y, RudeColor *color);
virtual void Blt(long x, long y, RudeSurface *src, RudeRect *srcrect);
...
```

`RudeSurface` is an abstract class designed to be the basic 2D drawing surface primitive for games using the Rude Engine. The method `CreateSurface()` should be used to create a surface, this always return a drawing surface appropriate for the given platform. Several implementations of `RudeSurface` are available, all transparent to the programmer:

1. `RudeSurfaceGapi`, a GapiDraw surface implementation usable on Windows Desktop, Windows Mobile SmartPhone, Windows Mobile Pocket PC and Symbian ARMI targets.

2. `RudeSurfaceSymbian`, a surface implementation for Symbian emulator and Symbian ARMI targets.

3. `RudeSurfacePalm`, a surface implementation for Palm devices.

More *RudeSurface* implementations may be available in the future, and developers are of course welcome to write their own implementations.

4.3 RudeSurface3D

```
void SetSurface(RudeSurface *surface);

void GrlDraw3DLine(RudeVertex *a, RudeVertex *b,
RudeColor *color, bool clip = false);

void GrlSetViewport(int top, int left, int bottom, int right);
void GrlOrtho(float ox, float oy, float oz, float w, float h, float d);
void GrlPerspective(float kk);
void GrlLookAt(float eyex, float eyey, float eyez,
float vx, float vy, float vz,
float upx, float upy, float upz);
void GrlLoadIdentity();

void GrlTranslate(float x, float y, float z);
void GrlScale(float sx, float sy, float sz);
void GrlRotate(float degrees, float ax, float ay, float az);
void GrlRotateView(float degrees, float ax, float ay, float az);
void GrlTranslateView(float x, float y, float z);
void GrlApplyTransformation(RudeVertex *dstVert, RudeVertex *sv);
void GrlProject(RudeScreenVertex *dstVert, RudeVertex *sv);
void GrlClipLine(RudeScreenVertex *a, RudeScreenVertex *b);
bool GrlZClip(RudeScreenVertex *a);
...
```

`RudeSurface3D` is the heart of 3D rendering in Rude Engine. `RudeSurface3D` is initialized from a `RudeSurface` object, and drawn using an OpenGL-style API.

4.4 RudeRect

```
long top;
long left;
long bottom;
long right;
```

4.5 RudeColor

```
int r;
int g;
int b;

void SetColor(int r, int g, int b);
void SetColor(RudeColor *c);
```

4.6 RudeVertex

```
float x, y, z;
```

4.7 RudeScreenVertex

```
long x, y;
float z;
```

4.8 RudeKey

`RudeKey` is an enumerated type that can hold any of the following values:

```
kRudeKeyUnknown = 0,
kRudeKeyUp, kRudeKeyDown, kRudeKeyLeft, kRudeKeyRight,
kRudeKeyStart, kRudeKeyA, kRudeKeyB, kRudeKeyC,
kRudeKey0, kRudeKey1, kRudeKey2, kRudeKey3, kRudeKey4, kRudeKey5,
kRudeKey6, kRudeKey7, kRudeKey8, kRudeKey9
```

4.9 RudeMath

```
static int Rand();
static float InvSqrt(float x);
static float Cos(float x);
static float Sin(float x);
static float ATan2f(float x, float y);
```

Many commonly used mathematics functions are implemented different or not available on several mobile platforms. `RudeMath` serves as a wrapper class for these functions so that the same code can be used on all mobile platforms.

4.10 RudeRegistry

```
static RudeRegistry * CreateRegistry();
virtual long QueryByte(TCHAR *app, TCHAR *name,
void *buffer, long *buffersize);
virtual long SetByte(TCHAR *app, TCHAR *name,
void *buffer, long buffersize);
```

Saving game state in between runs of the game is challenging, especially across platforms. The `RudeRegistry` class solves this problem by creating a platform independent abstraction for saving key/value pairs.

4.11 RudeSound

```
static RudeSound * GetInstance();

void PlaySong(char *song);
void PlayWave(int num);
void Tick();
void Pause();
void Unpause();
void ToggleSound();
void SoundOn(bool soundon);
bool SoundOn();
void Shutdown();
```

`RudeSound` is a singleton class that creates a platform independent sound interface for your game.

4.12 RudeGlobals

```
static RudeGlobals * GetInstance();

RudeColor* GetWhite();
RudeColor* GetBlack();
RudeColor* GetRed();
RudeColor* GetGreen();
RudeColor* GetBlue();
```

`RudeGlobals` is a singleton storage class that holds constants useful to Rude Engine applications and other classes within Rude Engine.

4.13 RudeDebug

```
static RudeDebug * GetInstance();

void Render(RudeSurface *surface);
void Print(TCHAR *msg);
void Print(TCHAR *msg, TCHAR *arg);
void Print(TCHAR *msg, long arg);
void Print(TCHAR *msg, int arg);
void Write(TCHAR *msg);
void PurgeLog();
```

The `RudeDebug` class is a singleton class that serves as a platform independent debug log mechanism. If Rude Engine is compiled with `RUDE_DEBUG` defined, messages sent to `RudeDebug` are automatically printed to the screen and written to the file `log.txt` in the working directory of the game.

`Print()` formats log messages as `msg: arg`, and automatically calls `Write()` to write the message to the log file. Developers who only want debug information written to disk can call `Write()` directly.

It is not necessary for applications to call the `Render()` method if `RUDE_DEBUG` is defined.

`PurgeLog()` empties the file `log.txt`, and is called automatically upon initialization of `RudeDebug`.

5 Acknowledgments

The OOP concepts described in this paper come mostly from Dr. Timothy Budd's *An Introduction to Object-Oriented Programming* [1]. Dr Budd's book lays a solid foundation for understanding OOP principles in a language-independent manner that probably isn't available anywhere else. For the mobile development portions of this paper, thanks should be given to Douglas Beck (Digital Concepts), Johan Sanneblad (Viktoria Institute), and John Romero (Monkeystone), all of whom have assisted me to varying degrees in understanding the complexities of mobile application development.

References

[1] T. Budd, *An Introduction to Object-Oriented Programming.* Addison-Wesley, 2002.

[2] "Coding Idioms for Symbian OS for C++ Programmers," 2004. [Online]. Available: http://ncsp.forum.nokia.com/downloads/nokia/documents/