ChangeEngine Game Engine

By Cedric Wienold,
California Polytechnic Unversity,
San Luis Obispo, CA

Advised by Dr. Michael Haungs

June 11, 2011

Date Submitted:_____

Advisor:_____

Contents

Abstract

We propose to design a game engine which will provide the framework for fast, simple deployment of games. We analyzed several different game engine designs to allow programmers to work on game design rather than programming the structure of windows, graphics, sounds, et cetera, but allow them to program a wide range of 2D games. The programmer will handle the logic of each interaction on his own, but trivial functions such as drawing, collision detection, and input will be handled by this engine. The engine is available as a static library programmed in C++, which gives the user the power to program in any language that can implement such a library. A major feature of this engine is pluggable functions which can be implemented by the user, such as artificial intelligence.

1 Introduction

Game engines can be difficult to learn for a programmer who has just downloaded the libraries or source code for it. The main issues we see are that the programmer is expected to do a lot of work to get started: rudimentary operations like window creation and input detection can be hard to wrap one's mind around. Our goal is to look from a novice's point of view, with an engine which will hand-hold them through these operations until they feel able to extend my own engine's classes. We have no delusions of grandeur: If a programmer sufficiently experienced in game programming sees a need that we do not fulfill, he is free to overload it with his own code.

We also feel that the basic logic of the program is the responsibility of whoever is programming the game. Our engine provides functions which are necessary to most games, namely collision detection, but the implementer of the engine is responsible for what happens between the objects our engine provides.

2 Game Engine

The game engine supports two main ways of implementing its classes: managed and unmanaged.

2.1 Managed Engine

The managed engine provides memory management benefits and ease of use at the cost of flexibility to the user. Everything the programmer creates or instantiates will be done through the overarching game engine class.

2.1.1 Usage

1. `ChangeEngine* engine = ChangeEngine::Initiate();`

   This is the overarching game engine class, through which most of our operations will take place.

2. `engine->setWindowCaption("Test Window");`

   The programmer may wish to use a custom window title, as this is a windowed application. Full screen is not yet supported.

3. `engine->createWindow(screen_width,screen_height,bpp);`

   This sets up the game window with the given width, height, and color depth.

4. `engine->createLevel("Level1");`

   Scenes in this engine are split into "Levels", each of which contains several GameObject classes.

5. `engine->createGameObject("Level1","Object1");`

   This creates "Object1" and places it into "Level1" to be handled there.

6. `engine->attachImageToGameObject("Level1","Object1", "filename.png",tilewidth,tileheight);`

   This will attach an image to "Object1", also known as an 'avatar'. This is not a default operation for GameObjects, as it is likely the user will want to have an object with no image (for logical operations and such). "tilewidth" and "tileheight" are terms used in the context of tile sets, where multiple sprites of animation are contained in the same file. Sprite width and height are considered to be the same for all sprites in a tile set.

7. `engine->addAvatarState("Level1","Object1",spritecount);`

   This creates a state for the avatar and sets the number of sprites in that state. In the actual image file, each state is on a single row of tiles, and the sprite count is the number of sprites available on that row of animation. In this way, the tile set can have a variable number of frames of animation for each state.

   A state can be something as simple as the direction the object is facing, or an action it is taking, or both. The programmer must make sure to keep track of the order states are given, as they added from the top of the file down, and are indexed by integer starting from zero.

8. `engine->drawObject("Level1","Object1",state,frame);`

   This will draw "Object1" of "Level1" to the window with the given state and frame, as given above. Going outside the bounds of the number of states added, or the number of frames in that state, will result in undefined operation.

9. ... Game Operations ...

10. `ChangeEngine::Destroy();`

    This will make use of the game engine's internal memory management framework to remove all objects created so that the programmer need not worry about it.

2.2 Unmanaged Engine

The game engine provides the programmer with unrestricted access to each class contained therein. While the programmer must do his own memory management to prevent leaks, he is not restricted to our "Level" framework or even our own drawing functions. What follows is only an example of what can be done with the freedom of an unmanaged engine.

1. `ChangeEngine* engine = ChangeEngine::Initiate();`
2. `engine->setWindowCaption("Test Window");`
3. `engine->createWindow(screen_width,screen_height,bpp);`
4. `GameObject* object = new GameObject();`
5. `object->setWidth(tilewidth);`
6. `object->setHeight(tileheight);`
7. ... Game Operations ...
8. `object->drawImage(engine->getWindow(),state,frame);`
9. `SDL_Flip(engine->getWindow()->getScreen());`
10. ... Game Loop until Finished ...
11. `GameObject::Destroy(object);`
12. `engine->Destroy();`

We have made every attempt to expose useful variables in each object. One need only browse the available header files to understand each function. For example, ChangeEngine's GameWindow class contains an SDL_Surface called "screen", which we make available for use in SDL operations with

`getScreen()`

.

Take note that the GameObject class also has its own static destruction function. In an unmanaged engine, it will be the programmer's responsibility to locate all applicable class destruction functions and use them.

2.3 Object Controllers

This engine uses the concept of "controllers" to handle objects that need to be moved for whatever reason. Every object has a private GameController instance, which is null at first, since many objects may not require being controlled. The controller depends on using a managed engine, or an effective use of GameLevels. The game engine provides

```
actObjects(GameLevel*)
```

as an easy way to force all objects to be controlled, and this can be done every loop of the game engine.

The GameController class is an abstract class. In order for the programmer to control an object, he must make a new class which inherits GameController and write the

```
act(GameLevel*,GameObject*,EventListener*)
```

function himself. The act() function will take in three variables: the level the object is on, the object the controller acts on, and an event listener. Not following these standards for the variables will result in wholly undefined behavior.

2.3.1 Artificial Intelligence

The implementation of this function can be as simple as moving the object back an forth on its own, but its access to the level makes it aware of all objects in the field, so that the programmer is able to use artificial intelligence to make the object decide its actions based on its environment.

2.3.2 User Interaction

The controller object also allows the programmer to have the user interact directly with the object. The reason the EventListener is included in the function is to make the object aware of events such as keypresses or mouse clicks and act accordingly.

3 Design and Implementation

ChangeEngine will fulfill the following software requirements:

1. General Requirements
   (a) Single library implementing all functions for a rudimentary game
      i. Engine Class
      ii. Window Container
      iii. Level Container
      iv. Game Object Container
      v. Game Avatar (Tilemap Container)
      vi. Event Listener
      vii. Object Controller
   (b) Extensible classes for programmers to "plug in" their own logic
      i. Object Controller (Artificial Intelligence or User Input)
      ii. Event Handler

4 Tools and Teamwork

This project will primarily be programmed by Cedric Wienold using various programming environments supporting C++ code. Teamwork will be limited to interviews with industry professionals concerning algorithms and class organization advice.

5 Analysis

Testing this engine is an important way of determining if its current programming is at all useful for programmers.

To test the engine's managed object framework, I would run through a loop of progressively more objects (tens, hundreds, thousands, tens of thousands, etc.) and log the amount of time the game engine takes to create the objects, use the objects functions (such as attaching images and using the controllers on every loop), and destroy the objects.

6 Related Work

7 Conclusions

The programmer must be careful to ensure his usage of this class's available functions is logical.

7.1 ChangeEngine

`pollEvents()`

should only be called once per loop to ensure that multiple functions depending on the same event are not deprived of it because they did not act on the event quickly enough.

`createWindow(int w, int h, int bpp)`

should only be used with sensible, non-negative values.

`createLevel()`

and

`createGameObject`

have protections against being called with same-named values repeatedly. Keep in mind that if debug mode is off, the programmer will not be informed of a failure unless he is specifically checking for it.

`addAvatarState()`

should be used on a row-by-row basis for tile sets.

`drawObject()`

will not actually output an object to the screen.

`drawFinish()`

is necessary to flip the backbuffer to the front, and should only be called once every loop to prevent flickering.

8 Future Improvements

This project will remain open source for any programmer to extend and improve upon it. ChangeEngine will maintain a pluggable interface in many ways so that programmers need only design their own functions and pass them into the engine if they wish to use them. Artificial intelligence, collision detection, and input detection are all areas which programmers can design their own plug-ins for.

As far as future improvements to the game engine go, the following is a noncomprehensive list:

1. Change object references from strings to integers, to make loop-based instantiation easier.
2. Use several debug modes to change terminal output.
3. Continue building more specific error codes for all classes.
4. Create single "draw" function for a level which will draw all managed objects.

5. Handle timers or threads in ChangeEngine so the user need not be concerned with it.

6. Include a pointer to a CollisionDetector class in every level, and implement said class.

7. Fix the problem where the screen will not update its contents despite actual changes in Object coordinates.

8. Improve documentation of functions not in ChangeEngine class, to improve support for non-managed implementations.

9. Put level and object destructor access in ChangeEngine class.

10. Expose individual GameLevel and GameObject instances to the user via the ChangeEngine class.

A Research

A.1 Collision Detection

In an interview with Cal Poly Alumni Computer Science alumni Daniel Nutting, methods of collision detection employing best possible complexity in average cases were outlined, with worst possible complexity in edge cases being sacraficed. This method will be called the "Grid Collision Detection" method.

A.1.1 Grid Collision Detection

Theory: Truly basic collision detection entails each object checking against all other objects for collision, and doing the same for every object. This leads to an average $O(n^2)$.

The most apparent problem is repeated checks. One solution can be a stack method. For each object, push onto stack. Check collision with everything behind it.

The next issue is restricting the number of objects to check on. If objects are not close to each other, there is no point in checking. This is where the grid comes in. For a set of objects, calculate the average X and Y coordinates, and split the field there into 4 quadrants. Do this more as is necessary for the number of objects.

B Source Code

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla ac felis nibh. Maecenas vel risus erat, non convallis risus. Vivamus congue suscipit porta. Fusce cursus lobortis metus et rhoncus. Nulla in ligula eget felis vestibulum iaculis. Nullam ac enim at lorem iaculis accumsan. Nullam at diam a turpis blandit tincidunt ut vitae ipsum. Integer quis enim eget est porta rhoncus. Vivamus commodo, lacus nec laoreet pretium, nunc lectus tincidunt mauris, blandit lobortis erat ligula pretium quam. Mauris et libero nec felis tempor aliquam. Maecenas lacus ante, fringilla eget convallis et, fringilla a dui. Nam bibendum, elit porttitor blandit imperdiet, est magna fringilla diam, eu accumsan ante nisl at metus. Cras non gravida quam. Nulla nec lorem lorem. Proin et massa mauris. Aenean non metus orci. Maecenas porta purus et ligula consequat tempus.