Practical Assignment 2

Game engine basics

February 20, 2012

In this assignment, you will prepare a basic game engine. A VS template solution is available on the course website and you must use it as a starting point for this assignment. In that solution you will find, among others, the following files:

- `main.cpp` The program is a Win32 application (not a console application), so it makes use of the window-based functionalities. The main file consists then mostly of a `WndProc` function to handle the messages and the `WinMain` function, entry point of the program. You will notably notice that the latter creates one object of type `GameEngine`.
- `GameEngine.h` and `GameEngine.cpp` This class holds your game engine and all its functionalities. You will have to modify largely the code of that class in order to communicate between the different components of the engine. For now, it only contains functions communicating with the graphics component, making calls to a `RendererOpenGL` object.
- `RendererOpenGL.h` and `RendererOpenGL.cpp` This class holds the functions managing the OpenGL specific calls. It allows for the creation of an OpenGL rendering window, its resizing, a sample scene drawing *etc.*

If you run the program, you will see a window with two 3D objects (colorful pyramid and cube) that are rotating. Press `F1` to toggle to fullscreen, press `Q` or `Esc` to quit. You can also resize the window by dragging one of its corner.

The assignment consists of 5 tasks having the purpose of improving that game engine template. You do not have to implement them in the following order but it is recommended.

Task 1: Renderer interface

The rendering of a game is a major factor for its success. You can program the best game logic ever, if the rendering is poorly done or inappropriate, the game is doomed to fail. Rendering systems are not all equivalent and therefore it is useful to design an engine that is capable of working properly whatever rendering system is used. By doing so your game engine can be used, without any modification, to program very different games with very different visual aspects. And indeed after all, the way you render a game entity (the player, an item, the sky *etc.*) does not influence the properties of that entity (global position in the world, number of remaining lives or amno *etc.*). So we can imagine an interface between the renderer specific calls and the game entities that need to be drawn on the screen.
The first task of this assignment is to design such interface (using an abstract base class `Renderer`). To start in the right direction you can imagine that you would like the possibility of selecting -at start up- between a Direct3D based rendering system and an OpenGL one. You will need an interface between the game engine (calls independent from the selected renderer, *e.g.* you will always need a function to create/kill the window, to resize it, to switch to fullscreen *etc.*) and the renderer system itself. Inside that `Renderer` abstract class, you will also have to create a bunch of functions to draw primitive objects, perform transformations, change the current color *etc.* The existing `RendererOpenGL` class will have to be modified to fit those functionalities. Calls to OpenGL specific functions are only allowed in the body of `RendererOpenGL`. The `GameEngine` class will of course have to be slightly modified to match the functions of the abstract class.

Task 2: Creation of the engine components

As you can see, the GameEngine instance is created at the beginning of the `WinMain` function and deleted at the end. This makes sense but it does not ensure that only one instance of GameEngine can be created in the program. It would be better if we could ensure that it is actually impossible to create a second one and that we can access the unique instance easily from anywhere in the code.
The second task of this assignment is to design the GameEngine class so that this property is fulfilled. Later you will apply this design to every appropriate engine components (*e.g.* you also want unique managers of resources, sounds, inputs ...).
You have to alter the GameEngine class and adapt its creation/access/destruction in the main file.

Task 3: Update and Draw

For now the program loop is done in the `WinMain` function by the control structure **while**(!done). It basically consists of the following statements: if a message (event) needs to be treated then it is solved, else the window is re-drawn if active. You might have noticed that the game logic is not updated (only the two rotations hardcoded in the renderer that you have to remove).
The third task of this assignment is to design a game logic update and entity rendering mechanism. Each game entity that needs to be updated, have to be called by an `update` method. Be aware that an entity that needs to be drawn does not necessarily need to be updated (*e.g.* the ground or walls), and that an entity that needs to be updated does not necessarily need to be drawn (*e.g.* small bullets). Therefore you will need two different mechanisms for that. An entity will inherit from a specific abstract class (`Updatable`) if it needs to be updated and another one (`Drawable`) if it needs to be drawn. The primary characteristic of the class `Updatable` will be a function `update`, and the primary characteristic of the class `Drawable` will be a function `draw`.
In addition, each entity should be updated according to a specific frequency (in Hz). Indeed for example you do not need to update the AI of an enemy at maximal frame rate, you often prefer to save that time to do something more critical. You also want your game to run at the same *speed* whatever computer you use, you do not want to see your enemies running and shooting at excessive velocity on a faster computer than the one you developed on... This can be managed by making sure that an entity is updated only after a predefined amount of time. And the other hand, the rendering is performed as fast as possible for the entities that have to be drawn (the time saved in the update is then used for a better rendering, special effects *etc.*).
The calls to the `draw` and `update` functions of each appropriate entity are done in the `GameEngine` class. So you will have to register the entities into two containers (do not forget the possibility to unregister them). The registration (resp. unregistration) is done at the creation (resp. destruction) of the object. It can so be done in the abstract classes. The calls to the `update` and `draw` functions of the game engine are done at each frame in the main loop. These functions scan the two containers and call the `update` and `draw` functions on each entity.
Once this mechanism is realized, create few objects that are 'updatable', 'drawable' or both. Try out different update and drawing codes as well as different update frequencies.

Task 4: Game state

When you play a game, you are not always in a pure playing mode. Sometimes you need to pause the game, for example to change an option in a menu, or sometimes the game is playing back a video or a in-game cinematic. An easy way of implementing this is to use game states (*e.g.* `Playing, InMenu, Video, Cinematic, Pause`). Depending of the current state, the game loop will execute a different block of statements.
The fourth task is to implement game states in your engine. Select relevant states (at least `Playing` and `Pause`) and modify the `GameEngine` class in accordance. Try them out, at least by toggling to a pause mode when pressing the key `'P'` (binding done in `main.cpp`).

Task 5: Input manager

In this engine template the management of keyboard and mouse inputs is done in the `WndProc` procedure using an array **bool** keys[256]. To deal with inputs more efficiently, an `InputManager` class will centralize the messages related to inputs and keep track of the current keyboard and mouse states (accessible therefore by polling). The manager will define functions such as `mousePosition, isKeyDown` and `mouseLeftButtonDown`. For that the `uMsg, wParam` and `lParam` parameters of `WndProc` need to be forwarded to the manager at each new message. You will design a function `InputManager::update` taking these values as arguments and called from `WndProc` for each message.
In addition, it is often useful to allow objects to be notified that an input event occurred. They can then take action according to the input. The object that will notify these listeners is the input manager. All listeners will have to implement specific functions (at least `keyPressed` and `mouseMoved`). Keyboard and mouse events/listeners are treated separately using two abstract classes `KeyboardListener` and `MouseListener`.
The last task of this assignment is to implement both the input manager and the input listener mechanism. The `InputManager` class will manage the (un)registration of the keyboard and mouse listeners, and will provide with polling functions (accessed by the listeners when notified). To be a listener, a class just has to inherit from the abstract class(es) and implement the relevant function(s): (at least) `keyPressed` for the keyboard listener and `mouseMoved` for the mouse listener. For instance the `GameEngine` class is a keyboard listener (and probably at some point a mouse listener as well) allowing to pausing, switching to fullscreen and exiting the program by monitoring the keyboard events (not done in the main function anymore). Finally you have to make sure that in the main file, you do not use **bool** keys[256] anymore (moved to `InputManager`), you do not update key states in the `WndProc`, you create only one input manager, and you do not poll key states in `WinMain`.

More

If you have extra time, you can improve your game engine by implementing the following optional tasks.

- On the msdn website, you will find the exhaustive list of possible input messages:
  Keyboard: http://msdn.microsoft.com/en-us/library/windows/desktop/ff468861(v=vs.85).aspx
  Mouse: http://msdn.microsoft.com/en-us/library/windows/desktop/ff468877(v=vs.85).aspx
  You can implement, in your `InputManager`, as much notifications as you find relevant for a game engine. This will give additional features for your listeners.

- After completion of task 1, the `Renderer` gives you access to a bunch of functions to draw primitive objects, perform transformations, change the current color *etc.* Try to add as much functionalities as you find relevant for a game engine. You can get inspiration from the OpenGL documentation:
  OpenGL GL: http://msdn.microsoft.com/en-us/library/dd374211(v=vs.85).aspx
  OpenGL GLU: http://msdn.microsoft.com/en-us/library/dd374158(v=vs.85).aspx
  OpenGL GLUT: http://www.opengl.org/resources/libraries/glut/spec3/spec3.html

- After completion of task 1, you have a `Renderer` class that can be used to perform the rendering using different systems. Do it using Direct3D through a class `RendererDirect3D` that inherits from `Renderer`.
  The DirectX SDK can be downloaded here: http://msdn.microsoft.com/en-us/directx/aa937788
  Documentation on Direct3D can be found here: http://msdn.microsoft.com/en-us/directx/bb896684
  This optional task is rather complex (it requires a good knowledge of Windows programming).

Submitting your work

The submission deadline is Sunday March 4 at 11:59pm. To send us your work, you will use the CS-UU submission system available at http://www.cs.uu.nl/docs/submit. Please DO NOT send your

assignment by email.

Design your code in an object-oriented fashion. So, use classes and inheritance where appropriate. Please include sufficient comments to the code so that at least the role of each function and class in your program is clear. Include as much error checking as appropriate. Double check that there is no memory leak. Make sure that you do not copy objects when not necessary.

If your project does not compile and run in both debug and release mode, there will be no grading. The fulfillment of the tasks and the design of your classes will be the major grading factors. Beside that, the resulting game engine mechanics, the quality and organization of your program, an adequate number of comments and the implementation of the optional tasks will be taken into account.

Please make sure to *clean* your solution (no temporary files and no .sdf). The assignment consists of the solution file (.sln), the project files (.vcxproj, .ico, *etc.*), the source files (.cpp and .h) and the external libraries (Libs folder). Create a .zip or .rar archive of your assignment files and upload it using the submission system. After a successful submission you should receive a confirmation email in your student email account. If not, please contact Jeroen Fokker.