Practical Assignment 3

Game engine components

March 5, 2012

In this assignment, you will improve your game engine by implementing additional functionalities. No VS solution are provided, you start from the current state of your own engine.

The assignment consists of 5 tasks. You do not have to implement them in the following order but it is recommended.

Task 1: Game vs. game engine

During assignment 2, to test your game engine, you had to create game entities. You had the choice of doing so by creating objects either in the `WinMain` function, the constructor of `GameEngine`, a dedicated function in the `GameEngine` class or any equivalent way. This is not a proper way of developing a game as you do not want to change the core of your game engine while developing the game logic (and vice-versa). You need to separate what makes the game engine and what makes the game. To do so, you will create a new class `MyGame` (or `Game` or any name for the game you would like to develop in assignment 4 if you have already decided). This class will inherit from `GameEngine` to have access to every functionalities of your game engine. This class will contain all the game related data and functions, while `GameEngine` will contain all data and functions that could be reused to develop another game. `GameEngine` has now two pure virtual functions `void createScene()` and `void deleteScene()` that needs to be overridden in `MyGame`. In these functions you will create (resp. delete) the game entities needed in the game. You can already put in them the few lines of code that you used to test your engine during assignment 2.

The derived class `MyGame` inherits the functionalities of `GameEngine`. Among them, some member functions will be very useful to override. For those, you will have to declare them virtual in the base class and probably call the base class version (as not empty) from the derived class.

Finally do not forget in the main file that you now create an instance of `MyGame`. You will then have to modify the access to the creation of the `GameEngine` instance.

Try out the class `MyGame` by testing different pieces of code in `createScene`.

Task 2: Scene management

So far, you have created independent updatable and drawable entities in `MyGame::createScene`. You did it either 'on the air' (if the pointers to the objects are not stored, their lifetime being then the same as the program) or by using a container (making the destruction of the objects possible in `MyGame::deleteScene`). But usually in a virtual world, entities are organized so that a transformation applied on an object is transferred to every object attached to it. For example, a wheel of a car has its own local rolling orientation but if the entire car is rolled upside down the wheel will be as well. Another example can be seen in human body modeling, if you rotate the thigh of a virtual character you want its calf to follow that same rotation. This is done by organizing entities in a structure called scene graph.

Task 2.1: Scene graph

Every entity is attached to a node of the scene graph. Each node consists of entities and other nodes. No loops are allowed, the scene graph is a tree-like structure not a graph-like structure (in opposition to what its name could suggest). The local transformation of a node affects both the entities attached to that node and all its children. Updating/Rendering the scene consists then in scanning the scene graph from the root to the leafs and calling the `update`/`draw` functions on each entity.

You have to implement a scene graph for your game engine. You will find in Appendix the declaration of the two new classes you need to implement: `SceneManager` and `SceneNode`. Adjustments to these classes are allowed if necessary. The scene manager is used as entry point to the scene graph and to perform simple tasks such as getting a specific entity, starting the update/rendering or cleaning the scene. `SceneNode` is the class implementing the tree-like structure of nodes and entities. In the following, `GameEntity` is the abstract base class for any updatable and drawable entities. `SceneNode` allows for the manipulation of `GameEntity` objects attached to the node, of its children, of its position/orientation and of its update and rendering.

Your entities have now a name (`string`) that can be used to search for an entity in the scene graph. They also store a pointer to the node they are attached to. When an entity needs to update its location in the world, you will use the functions `getPosition` and `setPosition` on its node (and the equivalent functions for the orientation). It means that if multiple entities are attached to the same node, they will all move as soon as one of them changes the position/orientation of the node. If they should not, then you need to create dedicated nodes for those entities (using the `createChildSceneNode` function). The `draw` function of a specific entity now performs only local rendering calls (such as changing the color, drawing a cube *etc.*). The absolute position/orientation of the entity is given by the recursive calls to `enterRenderingSceneNode` (resp. `exitRenderingSceneNode`) done at the beginning (resp. end) of `renderSceneNode`. It also means that the `update` and `draw` functions of your abstract game entity class have to be virtual (if not already).

By using a scene graph the drawing procedures have to be called in a specific order (to perform the node transformations in the correct order). So you cannot anymore call the `draw` function on each entity in a 'random' order (*e.g.* order of registration). To draw the scene, `GameEngine::Draw` will now call the function `renderScene` of the scene manager, which will start the rendering procedure from the root node. The update procedure will follow the same principle (even if the calling order is usually less critical).

Finally, to create and to register game entities in your game (in `MyGame::createScene`) you will simply add them somewhere in the scene graph (optionally by creating a new node for it). The following piece of code creates two entities (one of type `Entity1` and one of type `Entity2`), the update frequency of the first one is 20, and the second is 2. The first entity is attached to a child of the root node while the second entity is attached to a child of the first entity node. It means that every transformation performed on the node of the first entity will affect the second entity.

```
GameEntity * entity1 = new Entity1(20,"entity1");
SceneNode * entity1SceneNode = SceneManager::Instance()->getRootSceneNode()->createChildSceneNode();
entity1SceneNode->addEntity(entity1);

GameEntity * entity2 = new Entity2(2,"entity2");
SceneNode * entity2SceneNode = entity1SceneNode->createChildSceneNode();
entity2SceneNode->addEntity(entity2);
```

If you wanted two independent entities, the second one can for example be attached to a child of the root node instead. Here is how you would define such node:

```
SceneNode * entity2SceneNode = SceneManager::Instance()->getRootSceneNode()->createChildSceneNode();
```

Try out the scene graph by creating entities and by attaching them to different nodes. You will probably have to change the update code (*e.g.* local from/to global transformations).

Task 2.2: Camera

A special entity in your game engine is the camera *i.e.* the position and orientation of the field of view. We will assume that you can have only one camera (one view point) in your game engine. You easily imagine that when the camera is moved, every entity moves accordingly. By using our scene graph, it means that the camera is placed on top of the scene graph. The camera is then a special entity (*e.g.* class `Camera`) attached to the root. To enforce that behavior you will ensure that no other entities can be attached to the root node.
Try out your camera by defining translation and rotation binded with user inputs (*e.g.* directional arrows and mouse displacements).

Task 3: Serialization

The ability to save and load the game is an essential feature in a game engine. Most game engines can for example save and load the player and level states, user profiles and preferences. In this assignment you will implement a saving/loading mechanism of the scene graph. You have to be able to save and load a complete graph *i.e.* the `SceneNode` and `GameEntity` objects. You should have reached the point where everything related to a specific game is created in `MyGame` and added to the scene graph. So saving the entire game state is equivalent to saving the scene graph state.

For that you will create a new abstract class `Serializable` containing, at least, the functions `write` and `read`. You should be able to call these functions with file-based streams and string-based streams. Every object that needs to be saved (at least the `SceneNode` and `GameEntity` objects) will then inherit from that class and override the two functions (see Appendix for `SceneNode`).

Every derived class of `GameEntity` will write and read its relevant specific data members (*e.g.* data members of `Entity1` and `Entity2`). In task 2.2 we have assumed that the camera is unique and the only entity attached to the root. You can then treat the camera separately as you do not need to write/read it. The root node of the scene graph does not need to be saved as well, you start the process with its children.

As it is very probable that you maintain a container of the `GameEntity` objects created in `MyGame::createScene`, you will need to update the container with the new pointers to the loaded entities. For that you will use an `AddressTranslator` class. This class should be able to add the relation between an old saved address and the newly created one of the same object: `void addAddress(void*, void*)`. It should also be able to translate an old address to the new one and to clear the translation table. Everywhere you stored pointers to `SceneNode` and `GameEntity` objects without being the owner of those pointers, you will use the translator to get the new pointers. You will gather the address translations into one function `void fixup()` called on each object having a pointer to objects it does not own. These functions are called after the reading of the entire scene graph.

To start the save/load mechanism, you will bind keys (*e.g.* `'K'` and `'L'`) to functions calling `write`/`read` on the root node the scene graph. Then each scene node will recursively write/read its own properties. At reading, you need to clean up the scene graph so that the new one (image of the old saved one) replaces the current one.

Try out the mechanism by saving the scene graph to a file, then loading it again (during the same execution and after a reboot of the engine). Try it out with destructions and creations of nodes and entities in-between the saving and loading.

Task 4: Plugin architecture

Mostly due to commercial reasons, it is important to be able to release new code without having to update the game executable. Imagine that you want to release a new extension for your game. All players are not necessarily willing to buy/install it. These players should have nothing to change (no new executable) when the extension is released. It means that both players (with and without extension) should be able to run the same executable but producing a different content. Even if it is difficult to foresee the game logic of an extension nobody knows anything about in advance, it is possible to prepare your game engine to dealt with such dynamic content. This mechanism can be implemented using a plugin architecture.

In this assignment you will implement such architecture to allow for the execution of a new piece of code to update the game logic of a `GameEntity` object. You can first start by creating a new derived class of `GameEntity` named for example `Entity3` (but you can also use classes that you already have). You will then add at least one object of this class to the scene graph (in `MyGame::createScene`). A pointer to the used plugin will be stored in that class and will be used to calculate the new position of the entity.

For that you have to design an abstract class `Plugin`. That class will contain at least the function `string getName()` to get the name of the plugin and the function `void getNewPosition(vector<double>&)` to update the position of the entity. Then you need to design a manager for your plugins, the class `PluginManager`. This class has to manage the loading and unloading of plugins that inherit from the class `Plugin`. You will find in Appendix the declaration of the class `PluginManager`. You are allowed to change it if necessary. To use the manager you first have to automatically insert, during its creation, the file names of all available plugins in the vector `_pluginsDLL`. You can assume that they all are in a specific folder and that the location of this folder will never change with relation to the executable file of your game. You can use the functions `FindFirstFile` and `FindNextFile` of Windows API to retrieve the full names of the plugin files (DLLs). Then before creating the scene you need to load the plugins by calling the function `LoadLibrary` on each DLL found. Next you need to create the plugin objects by first calling the function `GetProcAddress` to retrieve the pointer to the plugin function `createPlugin`, and then calling this latter.

When that architecture is ready, you can create a new empty project for your plugin. You can either add a new project to your game engine solution or create a new solution containing only that project. The plugin has to be compiled as a dynamic library (right click on the project `-> Properties -> General -> Configuration Type -> Dynamic Library (dll)`). The project will contain only one file (you can add more if you want to extend to functionalities of your plugin). This file will declare and implement a derived class of `Plugin`, including the function `getNewPosition` that will be used by the `Entity3` objects. Do not forget the line:

```
#define PLUGIN_EXPORT 1
```

at the beginning of the file and the following line at its end:

```
extern "C" PLUGINUSE Plugin* createPlugin(PluginManager& mgr) {return new NewUpdatePlugin(mgr);}
```

where `NewUpdatePlugin` is the derived class of `Plugin`.

Finally, at the creation of the scene, create a new `Entity3` object associated with the plugin. Try out your plugin architecture by changing the `getNewPosition` code. You should observe that you do not need to compile the game anymore, only the DLL.

Task 5: Basic engine tuning

When the feedback on your assignment 2 is available, please take into account as many comments as possible. Definitively solve the major errors and then tune your code to correct the smaller mistakes you made or improve on its flaws.

More

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

- Another special kind of entity in the scene graph is light. Indeed, lighting elements such as spotlights can also be placed in the scene graph. Imagine you have an indoor virtual scene with moving lamps, in order to see the lights moving along with their corresponding 3D objects they will be attached to the same node. This optional task consists in implementing lights (ambient, spot, *etc.*) in your scene manager.
- You may have notice that you create `GameEntity` objects at two locations in your code. The first one in `MyGame::createScene()` when you create your scene graph and attach the entities to the nodes. The second one in `SceneNode::read()` when you replace the current scene graph with the one saved. This is not a good programming design as the ownership of the created objects is ambiguous. To solve this, you will create a factory of `GameEntity` objects called `GameEntityFactory`. The game engine will then create the factory (with ID or by registration) and the `MyGame` and `SceneNode` classes will use it to create the objects.

Submitting your work

The submission deadline is Sunday March 25 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(s) (.sln), the project files (.vcxproj, .ico, *etc.*), the source files (.cpp and .h) and the external libraries (Libs and plugin folders). 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.

Appendix

SceneManager

```
class SceneManager {

public:
        static SceneManager * Instance();  // The public function to access the singleton
        static void destroy();       // Destruction of _instance
        static void create();        // Creation of _instance

public:

        Camera * getCamera() const;    // Get the camera
```

```
        void setCamera(Camera *);      // Set the camera

        SceneNode * getRootSceneNode() const;          // Get the scene node at the root of the scene graph
        GameEntity * getEntity(const std::string&) const;    // Get the entity from its name

        void updateScene();         // Update the entire scene
        void renderScene();         // Render the entire scene
        void clearScene();         // Empty the entire scene (nodes, camera and lights)

protected:
        SceneManager();       // Constructor
        ~SceneManager();      // Destructor
        static SceneManager* _instance;  // The one instance

        SceneNode * _rootSceneNode;     // The root scene node
        Camera * _camera;          // The camera

};
```

SceneNode

```
class SceneNode : public Serializable {

public:

        SceneNode(SceneNode *);                 // Constructor, parent node as parameter
        ~SceneNode();                      // Destructor

        void addEntity(GameEntity *);              // Add the entity to the scene node
        void removeEntity(GameEntity *);            // Remove the entity from the scene node
        GameEntity * getEntity(const std::string&) const;    // Search for an entity by its name
        void removeAllEntities();                // Remove all entities (no destruction)
        void deleteAllEntities();               // Delete all entities

        SceneNode * createChildSceneNode();           // Create and return a child of the scene node
        void removeChildSceneNode(SceneNode *);         // Remove a child node
        SceneNode * getParentSceneNode() const;         // Get the parent scene node
        void deleteAllChildrenSceneNode();           // Delete all children scene nodes (and remove entities)
        void deleteAllChildrenAndEntitiesSceneNode();       // Delete all children scene nodes and entities

        std::vector<double> getOrientation() const;       // Get the (local) orientation of the node
        void setOrientation(std::vector<double>);        // Set the (local) orientation of the node
        std::vector<double> getPosition() const;        // Get the (local) position of the node
        void setPosition(std::vector<double>);         // Set the (local) position of the node

        void updateSceneNode();                // Update the scene from the current node
        void renderSceneNode();                // Render the scene from the current node
        void enterRenderingSceneNode();             // Move into the node (push context, apply local transformation)
        void exitRenderingSceneNode();             // Move out of the node (pop context)

        bool write (std::ostream&) const;            // Write object to stream
        void read (std::istream&);                // Read object from stream

private:

        std::vector<SceneNode *> _children;           // The children of the scene node
        SceneNode * _parentSceneNode;               // The parent node (NULL if root)
        std::vector<GameEntity *> _entities;          // The entities attached to the scene node
```

```
        std::vector<double> _position;          // The current local position
        std::vector<double> _orientation;       // The current local orientation
};
```

PluginManager

```
class PluginManager {

public:

        static PluginManager * Instance();  // The public function to access the singleton

        void loadAllPlugins();      // Load all available plugins
        void unloadAllPlugins();    // Unload all loaded plugins

        Plugin* getPlugin(std::string); // Get a plugin by its name

private:
        PluginManager();        // Constructor
        ~PluginManager();       // Destructor

        HMODULE loadPlugin(std::string); // Load a plugin and return its handler

        static PluginManager * _instance;  // The one instance

        std::vector<std::string> _pluginsDLL; // The available plugins DLL names
        std::vector<HMODULE> _pluginsHandler; // The plugins handlers
        std::vector<Plugin *> _plugins;   // The plugins

};

#ifdef PLUGIN_EXPORT
        #define PLUGINUSE __declspec(dllexport) // Export in Plugin project
#else
        #define PLUGINUSE __declspec(dllimport) // Import in PluginManager class
#endif

extern "C" PLUGINUSE Plugin* createPlugin(PluginManager& mgr); // Create the plugin object

typedef Plugin* (*PCREATEFUNC)(PluginManager&);
```