# Lab 6 – Component System Part 1

In this lab we are going to implement a component based system. This will involve creating a Base Component that all other Components will inherit from, a Game Object which is used to manage all components and finally we create some Game Objects.

The instructions for this exercise will be truncated slightly, most of the code for this exercise is already in main.cpp and we will be moving things around to map onto the new components. If at any point you require assistance, please contact the Lab Demonstrator. Also there is a checklist you should fill out as you go!

## 1. Base Component

The base Component will be the blueprint for all other Components in the system. With this in mind we are going to specify a few methods which can be overridden by Child Components and finally a couple of variables which will be used by all Components.

### 1.1 Component Files

Add the following new files to the project

- Component.h
- Component.cpp

You can add files **by right clicking on the project** and selecting **Add->New Item**

Now add a **header guard** to **Componet.h**, for an example of this check out Texture.h.

### 1.2 Component Declaration

Add an empty class Declaration called **Component** to the header file, this should go inside the header guard and look something like this.

```
class Component
{
public:
protected:
private:
};
```

Now add an 'no parameter' constructor declaration to the public section

```
Component();
```

and now a **virtual** deconstructor

```
virtual ~Component();
```

The reason we mark the deconstructor as virtual is to allow the runtime to call the child classes deconstructor, this is pretty much a requirement if we are going to carry out some sort of inheritance. - http://stackoverflow.com/questions/461203/when-to-use-virtual-destructors

Again in the **public** section of the class, add the declarations for these functions

- **init** – no parameters, returns void, mark as **virtual**
- **update** – no parameters, returns void, mark as **virtual**
- **render** – no parameters, returns void marks as **virtual**
- **destroy** – no parameters, returns void, mark as **virtual**

The reason that these functions are marked virtual is that our Child Classes will be able to override.

In the **protected** section add the follow variables, you may need to add include statements for some variables

- **std::string m_Type;**
- **bool m_Active;**

Now shift back to the public section and add the following functions

- **getType** – no parameters, returns **const std::string&**
- **toggleActive** – no parameters, returns void
- **isActive** – no parameters, returns **bool**

### 1.3 Component Definition

Now we are going to define the class, open up **Component.cpp** and add an include statement for **Component.h.**

Define the constructor and deconstructor this should like the following

```
Component::Component()
{
}

Component::~Component()
{
}
```

Inside the constructor set the **m_Type** to “Component” and the **m_Active** to true.

Now we are going to implement the functions, this should look like the following

```
void Component::init(){}

void Component::update(){}

void Component::render(){}

void Component::destroy(){}

const std::string& getType()
{
        return m_Type;
}

bool Component::isActive()
{
        return m_Active;
}
```

Notice that all the virtual functions are empty, this usual for this sort of class.

Compile the application, if there any errors please contact the Lab Demonstrator.

## 2. Game Object

Now we are going to implement the Game Object class, this will be the container for the components.

### 2.1 Game Object Files

Add the header and source files both named **GameObject** to the project, add the header guard to the header file.

### 2.2 Game Object Declaration

Add the following to the header file, some of this may require include statements.

- Empty class with a public, protected and private keywords(see section 1.2)
- Add **init**, **update**, **render**, **destroy** functions, this should be exactly the same as the Component class(see section 1.2) to the public section, except that the **functions will not be marked virtual.**

- Add a **getName** function this should return a **const std::string&** to the public section
- Add a **setName** function, this should return a void and take in a **const std::string&** called **name** to the public section
- Add an **addComponent** function, this will take in a pointer to a **Component**(**Component * component**) and return void.
- Add a private variable called **m_Name** which is a **std::string**
- Add another private variable which is a **vector** of **Component** pointers called **m_Components**, this will look like the following
    - **vector<Component*> m_Components;**

**Now add the definition to the GameObject.cpp, remember to include the GameObject.h file to the top file.**

This should be fairly easy as most of the functions of will be iterating through the components and calling the equivalent function, for example the body of the **init** function will look like this

```
void GameObject::init()
{
	for (auto iter = m_Components.begin(); iter != m_Components.end(); iter++)
	{
		(*iter)->init();
	}
}
```

This is a pretty standard way of iterating through a vector, the **auto** keyword is used to tell the compiler to automatically to detect the type. We usually see this if the variable type may be unknown at compile time or just simple to save us typing, for example if we didn't use the auto keyword we would have to put the following into the first part of the for loop - **vector<Component*>::iterator iter=m_Components.begin()**

http://stackoverflow.com/questions/409348/iteration-over-vector-in-c

The **init**, **update** and the **render** function will look like the above, the **destroy** function will be slightly different. The reason for this is that we will be erasing items out of the vector as we iterate, this will mess with the order of the elements in the vector and invalidate the iterator.

The destroy function will look like the following

```
void GameObject::destory()
{
        auto iter=m_Components.begin();
        while(iter!=m_Components.end())
        {
                (*iter)->destroy();
                if ((*iter))
                {
                        delete (*iter);
                        (*iter)=NULL;
                        iter=m_Components.erase(iter);
                }
                else
                {
                        iter++;
                }
        }
}
```

The **addComponent** function will add a new component onto the vector, you would use the **push_back** function on the **m_Components** vector.

http://www.cplusplus.com/reference/vector/vector/push_back/

Now ensure that you implement the **get** and **set Name** functions, this is should very much like the component versions.

Compile the application, if there are any errors that you can not fix please contact the Lab Demonstrator.

## 3. Transform

The transform component represents a model transform, this will also contain variables for position, scale and rotation.

Like before add a header and source file, make sure you name both of them **Transform**. Again add an empty class to the header file, the class should inherit from **Component** class(you will also have to include the **Component** header file). For an example check out the following URL

http://www.cprogramming.com/tutorial/lesson20.html

Inside the empty class

- A public no parameter constructor
- A public deconstructor

- A private member variable called **m_Position** which is a **vec3**(you will have to include the GLM Maths & Transform includes)
- A private member variable called **m_Scale** which is a **vec3**
- A private member variable called **m_Rotation** which is a **vec3**
- A private member variable called **m_Model** which is a **mat4**
- A public **update** function(see 1.2)
- Public **get** and **set** methods for the **position**, **scale** and **rotation**
- Public get method for the **model** matrix

You should now switch to **Transform.cpp** and define the class, the only function that may take a bit more work is the **update** function. In this function you will use the various vec3's to build the Model matrix. Your update function should look like the following

```
void Transform::update()
{
        mat4 translate = glm::translate(translate, m_Position);
        mat4 scale = glm::scale(scale, m_Scale);

        mat4 rotationX = glm::rotate(rotationX, m_Rotation.x, vec3(1.0f, 0.0f, 0.0f));
        mat4 rotationY = glm::rotate(rotationY, m_Rotation.y, vec3(0.0f, 1.0f, 0.0f));
        mat4 rotationZ = glm::rotate(rotationZ, m_Rotation.z, vec3(0.0f, 0.0f, 1.0f));
        mat4 rotation = rotationX*rotationY*rotationZ;

        m_Model = translate*rotation*scale;
}
```

For more details on the above, have a look at the following link

http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/

You should also consider surrounding the above calculations with an "if statement" which uses the **m_Active** variable from the base component.

## 4. Mesh

The Mesh Class is the component that will be used to hold our visual elements, including Vertex Buffer Objects, Element Buffer Objects and Vertex Array Objects.

Again a new **header** and **source** file into the project, name both file **Mesh.** Add an empty **Mesh** class declaration to the header file, this class should inherit from **Component**. Remember you may need to add the includes for OpenGL(GLEW).

This should contain the following private member variables

- A GLuint for the VBO(**m_VBO**)**,** EBO(**m_EBO**) & VAO(**m_VAO**)
- An int for the vertex count(**m_VertexCount**) and an index cound(**m_IndexCount**)

For the public functions you should declare the following

- **init** function(see 1.2)
- **destroy** function(see 1.2)
- **copyVertexData** which takes in the following variables
  - **int** count
  - **int** stride
  - **void**** data
- **copyIndexData** which takes in the following variables
  - **int** count
  - **int** stride
  - **void**** data
- **Bind** function
- **Get** functions for the **Index** and **Vertex** count

Switch to the cpp file and define the class, most of the functionality has already been written in **main.cpp.**

- **init** function
  - copy the gen* functions from **createGeometry** function in **main.cpp**, ensure you rename all parameters of these functions to match onto the ones in the class. You should copy all vertex attribute pointers and vertex attribute array lines.
    You will probably want to copy all the **binds** as well.
- **destroy** function
  - copy the two **glDeleteBuffers** and the **glDeleteVertexArrays** from **cleanup** function in **main.cpp**
- **bind**
  - copy the binds from **createGeometry** to this function
- **copyVeretexData**
  - copy the bind for the VBO
  - copy the **glBufferData** for the VBO
    - change the 2nd parameter to be **count*****stride**
    - change the 3rd parameter to be **data**

    - set the **m_VertexCount** to be equal to **count**
- **copyIndexData**
    - copy the bind the EBO
    - copy the **glBufferData** for the EBO
        - change the 2nd parameter to be **count*stride**
        - change the 3rd parameter to be **data**
    - set the **m_IndexCount** to be equal to **count**
- Implement the get functions for the Index and Vertex

## 5. Material

The material class will be the component that handles the shaders , eventually textures and other variables which will govern how our objects 'look'

Add the Material class(again a source and a header file), this class should inherit from **Component**. Add the following variables to the private section.

- **GLuint m_ShaderProgram**;

Now add the following functions to the public sections

- **destroy**(see 1.2)
- **loadShader**, returns a **bool** and takes two **const strings&** for the filenames of the **vertex** and **fragment** shader
- **bind** return void
- **getUniformLocation** which takes in a **const string&** called name and returns a **GLint**

Switch to the source file and define the class.

- The **destroy** function, this should copy the **glDeleteProgram** from cleanup in **main.cpp**
- The **loadShader** should contain all the functionality from **createShader** , make sure you replace the vsPath and fsPath with the two strings that are passed in.
- The **bind** function should contain a call to **glUseProgram**(see the render function in **main.cpp**)
- The **getUniformLocation** function will be a thin wrapper around the call to **glGetUniformLocation** but istead of taking a hard code string in will take the string passed into the above function

## 6. Camera

The camera class will calculate the view and perspective matrix.

Again add a header and source file named **Camera** and add a blank **Camera** class, this should again inherit from **Component.**

Add the following variables to the private section of the class

- vec3 **m_Position**
- vec3 **m_LookAt**

- vec3 **m_Up**
- float **m_FOV**
- float **m_AspectRatio**
- float **m_NearClip**
- float **m_FarClip**
- mat4 **m_ViewMatrix**
- mat4 **m_ProjectionMatrix**

Add the following functions to the public section of the class

- **set** functions for all variables except **m_ViewMatrix** and **m_ProjectMatrix**
- **get** functions for **m_ViewMatrix** and **m_ProjectMatrix**
- **update** function(see 1.2)

Now you are ready to implement these functions, the above getter and setters are pretty simple. The **update** function will take the functionality from the **update** function in **main.cpp**