Instituto Superior Técnico

Cglib Tutorial

Creating a simple game using *cglib*

Carlos Martinho

Carolina Torres

2007

Table of Contents

Introduction 2

Step 0: Setting up the application in VS 2005 3

Step 1: Creating an empty application 5

Step 2: Creating a *cg::Entity* 8

Step 3: Draw the *box* on the screen 12

Step 4: Animating the *box* 16

Step 5: Create a mouse controlled *bat* 18

Step 6: Collision between *bat* and *box* 21

Step 7: Create more *box*es 23

Step 8: Create an arbitrary number of *box*es 24

Step 9: Using *cg::Group* to manage entities 28

Step 10: Application shutdown and memory leaks 30

Step 11: Adapting to window resizing 34

Glossary 38

References 39

Introduction

The following tutorial explains how to create a simple graphical application using *cglib*[1], an object oriented library that provides an abstraction over OpenGL[2] and GLUT[3].

Throughout this tutorial we will create a simple game by following some detailed steps. Each step will refer a specific part of the process of creating the game, while giving reference to *cglib* specific capabilities. Follows a summary of each step:

0. Creating a project in VS 2005; Description of the project's folder structure and essential files.
1. Creating an empty application displaying an empty window.
2. Creating a random *box*.
3. Drawing a *box* on screen; Learning about notifiers and listeners.
4. Animating a *box* with a constant linear velocity and configuring it to rebound on screen edges.
5. Creating a mouse controlled *bat*.
6. Implementing collision detection between the *bat* and the *box*es; Referencing a registered entity; Implementing the collision effect on the box.
7. Creating more *box*es.
8. Creating an arbitrary number of *box*es; Using a *cglib* listener to create HUD[4] (Head Up Display) like features.
9. The *cglib* group classes; Creating a "box manager".
10. Debugging memory leaks using VS2005; Handling keyboard events.
11. Handling window resize.

The final result will be a game with a mouse-controlled *bat* and random-moving *box*es that collide with each other.

The following figure shows an example of a possible final result:

Figure 1 - An example of a final result of this tutorial.

Step 0: Setting up the application in VS 2005

On this step we will setup the "tutorial" project in VS 2005. Also, we will describe the project's folder structure and which files are essential.

1. Extract the workspace cglib.dev to a local folder where you wish to work.

2. Copy the "template" project and rename it "tutorial" (note that this could be any other name, as long as you use that same name, rather than "tutorial", on the next steps).

3. Your folder file tree should now have an appearance similar to Figure 2.

4. In the "tutorial\vs2005" folder, rename "template.vcproj" to "tutorial.vcproj".

5. Open "tutorial.vcproj" in a text editor (e.g. notepad) and replace all occurrences of "template" by "tutorial". There should be only 2 occurrences of "template": at line 5 (Name="template") and at line 7 (RootNamespace="template"). Save the file.

6. Open the project solution in "cglib.dev\vs2005".

7. In VS2005, add a new project (as an existing project) to the *cglib* workspace. Select the "tutorial\vs2005\tutorial.vcproj" file.

8. From now on, add your source files to the project using the VS 2005 interface. All source files should be placed in the "tutorial\src" folder.

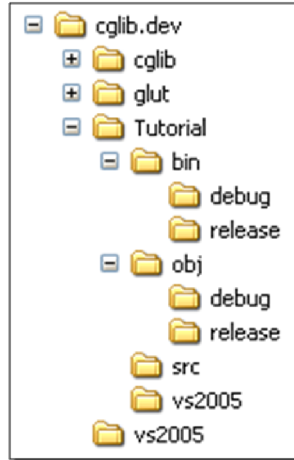


Figure 2 - *Cglib* tutorial file tree

To run the application we need to have an entry point. In any C++ application we need an *int main(int argc, char** argv)* method as an entry point. In VS 2005, create a new file, named "main.cpp", inside "Tutorial\Source Files". Inside this file, create the main method.

Figure 3 gives you an idea of how the solution in VS 2005 will look like after you finish:

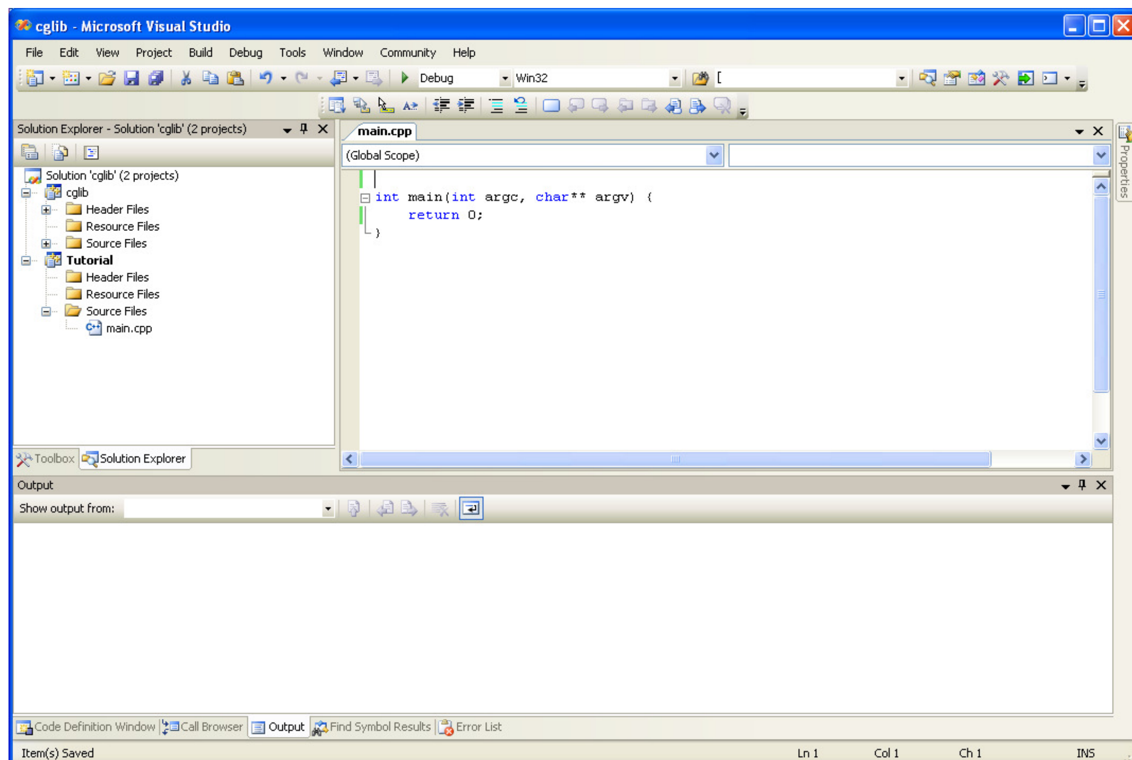
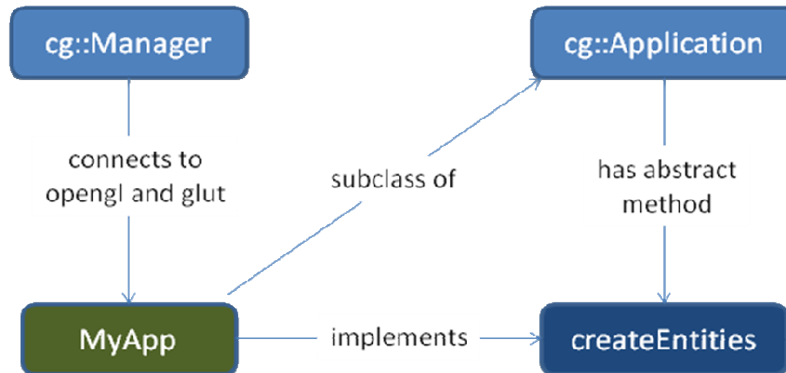


Figure 3 - The "cglib" and "Tutorial" projects in VS 2005

Step 1: Creating an empty application

In this step we will create an empty application displaying an empty window, using *cglib*. To do this, we create a subclass of the abstract class *cg::Application* and implement the abstract method *cg::Application::createEntities* (which will be used to add entities to the application, as we will see in the next step).

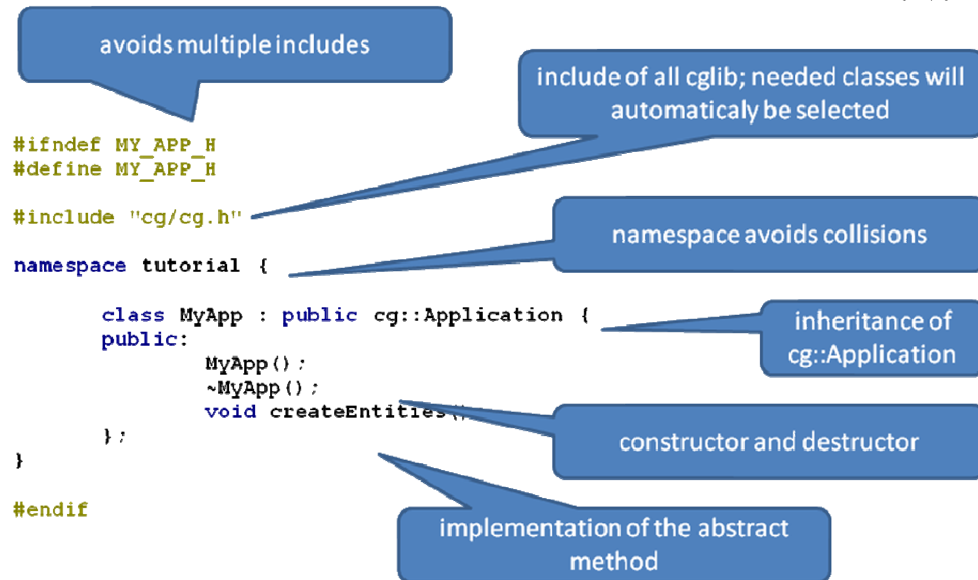


To create a subclass of *cg::Application* we need to add two new files to the "tutorial" project: "MyApp.cpp" and "MyApp.h". In C++, a .cpp file must always be accompanied by a header (.h) file. These header files are used to declare the variables and methods of the new class, which will be defined in the corresponding .cpp file. Defining only one class per file is suggested.

MyApp.h

In "MyApp.h" we will first define a macro MY_APP_H. This macro will be used for preventing multiple includes that could happen, for example, when a file includes another and this other file includes the first one, even if indirectly. We will also include the *cglib* header to allow us to freely use any *cglib* class.

In C++, all classes must be defined inside a namespace. A namespace is a context where each class name must be unique. Its best uses include aggregating classes with the same purpose. In this tutorial all classes will be defined in the same namespace named "tutorial". Furthermore, all *cglib* classes are part of the "cg" namespace.



As we can see in the example code, for *MyApp* to inherit from *cg::Application* we just need to refer it after the declaration of *MyApp* and before the definition of its methods and variables.

Every class must have a constructor and a destructor. The constructor may or may not receive arguments and we may have more than one constructor, as long as they receive a different number and/or type of arguments.

To implement the method *cg::Application::createEntities()* we must declare it inside *MyApp*.

MyApp.cpp

Since all needed files (such as "cg.h") are already included in "MyApp.h", in "MyApp.cpp" we need only to include its corresponding .h file. This a good practice and therefore will be used throughout the tutorial.

Again, we implement all the methods within the "tutorial" namespace.

MyApp.cpp

```
#include "MyApp.h"

namespace tutorial {

        MyApp::MyApp()  {
        }
        MyApp::~MyApp()  {
        }
        void MyApp::createEntities()  {
        }
}
```

includes only the correspondent header file

empty implementation of the abstract method

Now we must implement the declared constructor, destructor and abstract methods (i.e., *MyApp::MyApp()*, *MyApp::~MyApp()* and *MyApp::createEntities()*). In the example code we have an empty implementation of these methods, i.e., they are implemented but nothing happens when they are called.

main.cpp

To run a *cg::Application* all we have to do is call an instance of *cg::Manager* and its method *runApp*. To run the application we just created, we need to pass this method an instance of *MyApp* and the frame rate[5] we want it to have. The frame rate parameter allows us to adapt the application's performance to the machine's performance.

main.cpp

```
#include "cg/cg.h"
#include "MyApp.h"

int main(int argc, char** argv) {
        cg::Manager::instance()->runApp(new tutorial::MyApp(),60,argc,argv);
        return 0;
}
```

entry point of the application

connection of *MyApp* to OpenGL and GLUT via cglib

As you can observe in Figure 4 this step results in a running application showing an empty window.

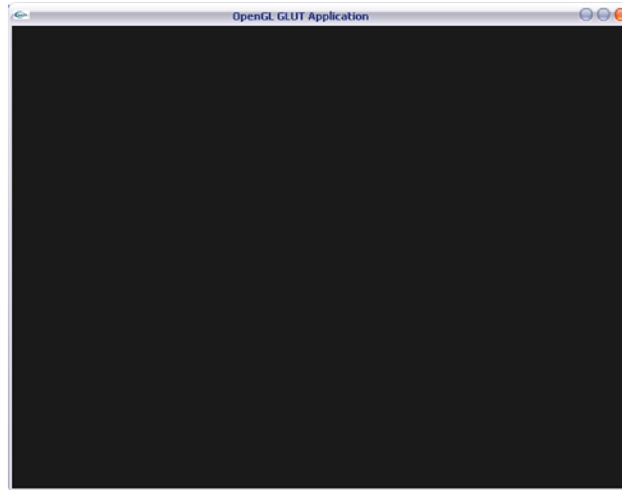
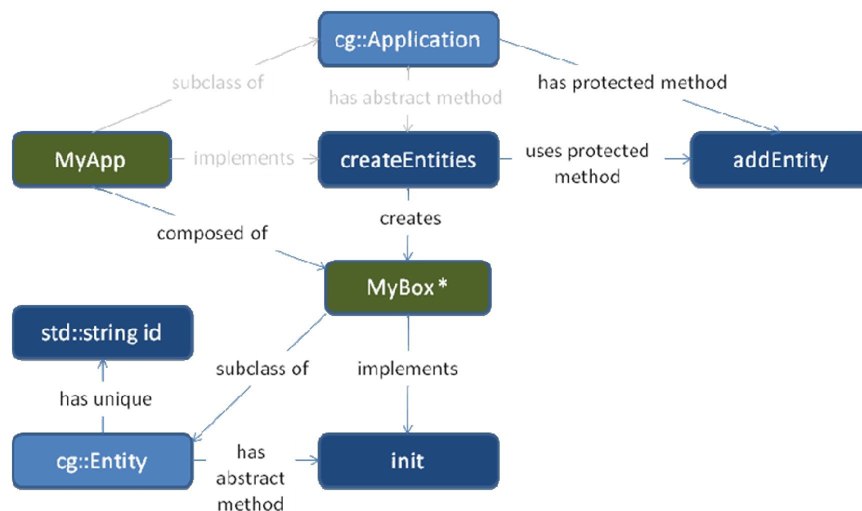


Figure 4 – Result of Step 1

Summary: We created an empty application by creating a class *MyApp* that inherits from *cg::Application*.

Step 2: Creating a *cg::Entity*

In this step, we will learn how to create a random *box* and add it to *MyApp*.



Due to the fact that *MyApp* inherits from *cg::Application*, it also inherits its methods. These methods can be rewritten by *MyApp*. By inheriting the constructor method *cg::Application(const std::string property_file)*, we are now able to load a configuration file when starting our application. This allows us to define general values in that file, such as the size of a particular entity[6], instead of hard-coding it. The configuration file should be placed in the "tutorial/vs2005" folder.

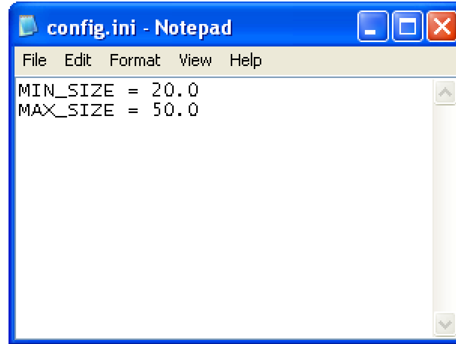
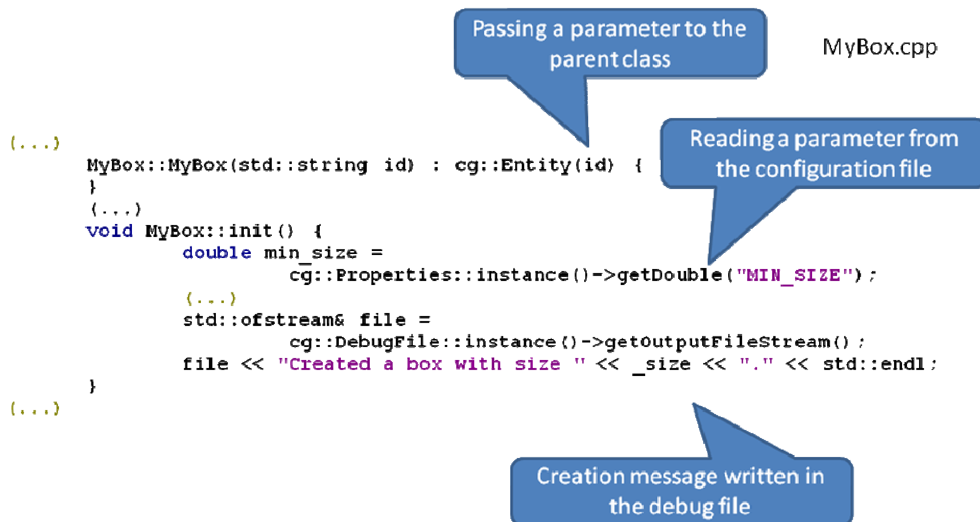


Figure 5 - The "config.ini" configuration file

In this example we created a file named "config.ini" and it has the definition for a MIN_SIZE and a MAX_SIZE, which we will use ahead. We will be able to access the values in this configuration file using the methods in the *cg::Properties* singleton[7] class (ex: *cg::Properties::getDouble(const std::string& name)*).



The *cg::Properties::getDouble(const std::string& name)* method receives a parameter name, searches the "config.ini" file for that parameter and converts it into a double type.

Another singleton class in *cglib* is *cg::DebugFile*. By keeping in a file every exception thrown, it facilitates debugging our application. We may also instruct the program to write other useful information to that file. To do so, we only need to use the *cg::DebugFile* class methods *write*, *writeLine*, *newLine* and *writeException*. For example, in the "MyBox.cpp" code shown above you can see the size of the *box* we just created being written in the debug file. The debug file is named "log.txt" and is located in the "tutorial/vs2005" folder.

As said before, we still have no entities in our application. To do that, we need to create a new class, *MyBox*, inheriting from *cg::Entity*. In the constructor we must provide an entity identifier, which is necessary to the parent class, *cg::Entity*. All entities can be accessed through this identifier, so it must be

unique. We must also implement the abstract method *cg::Entity::init*, which is where the entity is given its initial values.

MyBox.h

```
#ifndef MY_BOX_H
#define MY_BOX_H

#include <string>
#include "cg/cg.h"

namespace tutorial {

	class MyBox : public cg::Entity {
	private:
		cg::Vector2d _position, _size;
		cg::Vector3d _color;
		double _winWidth, _winHeight;

		double randomBetween(double min, double max);

	public:
		MyBox(std::string id);
		~MyBox();
		void init();
	};
}
#endif
```


The *cg::Vector2d* class allows us to perform vector calculus in a simple way. For example, if we wish to determine the distance between two vectors, v0 and v1, we need only to subtract one from the other and calculate the length of the resulting vector:

cg::Vector2d v0, v1;
v2 = v0 – v1;
distance = v2.length();

Now that we have an entity, we must add it to the application. To do that we should use the *cg::Application::addEntity* method. Entities should inherit from *cg::Entity* so that they can be added to the application using this method. The appropriate place to add entities will be in *MyApp::createEntities*. In the example code below you can see how a *MyBox* entity called *Box1* is being added to the application.

MyApp.cpp

```
#include "MyApp.h"

namespace tutorial {

        MyApp::MyApp()  : cg::Application("config.ini")  {
        }
        MyApp::~MyApp()  {
        }
        void MyApp::createEntities()  {
                addEntity(new MyBox("Box1"));
        }
}
```

Loading a configuration file

Creating a new *MyBox* entity

As a result, a *MyBox* entity will be created. We can verify this by opening the"log.txt" file inside "tutorial/vs2005". However, we get the same output we had in Step 1. The reason is that, although the *box* is being created, it is not being drawn. We will solve this issue in the next step.

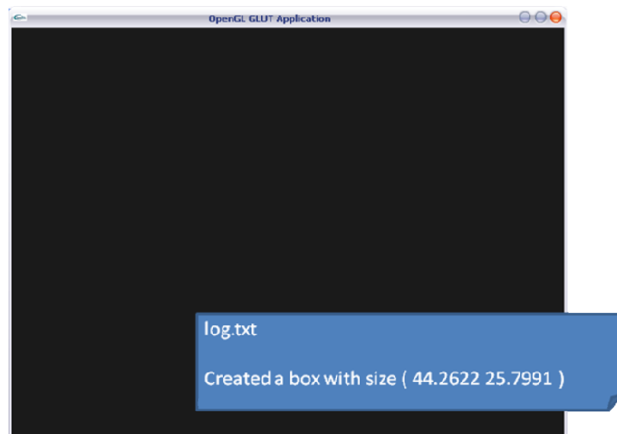


Figure 6 - Result of Step 1

Summary: We created a new entity, *MyBox*, by creating a class that inherits from *cg::Entity*. *MyApp* uses *MyBox* the same way as *cg::Application* uses *cg::Entity*. We also learned about the *cg::Properties* class that allows us to receive parameters defined in an external file, and about the *cg::Debug* class that allows us to write in an external file any useful information to the application.

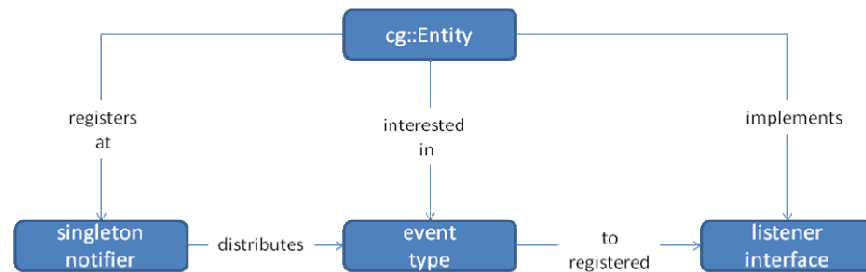
Step 3: Draw the *box* on the screen

In this step we will draw *MyBox* on screen. Also, we will learn about notifiers and listeners.

Listeners and Notifiers

To help in event registration, distribution and management, *cglib* provides several listeners through interface classes - *cg::IDebugListener, cg::IDrawListener, cg::IDrawOverlayListener, cg::IKeyboardEventListener, cg::IMouseEventListener, cg::IReshapeEventListener* and *cg::IUpdateListener*. When a class implements one of these listeners, it will automatically be registered in a notifier class. The notifier class will send warning to the listener whenever a relevant event occurs. For example, if an entity wishes to receive keyboard events, the entity's class must implement *cg::IKeyboardEventListener*.

Only one notifier of each interface class exists in the application and these notifiers are automatically handled by *cg::Application*.



Events

We have two sorts of events: "synchronous" and "asynchronous". Synchronous events – *update*, *draw*, *drawOverlay* and *debug* – are executed in a given order. Asynchronous events - *keyboardEvent*, *mouseEvent* and *reshapeEvent* – may occur at anytime during the application runtime.

The execution cycle of the synchronous listeners is shown in Figure 7.

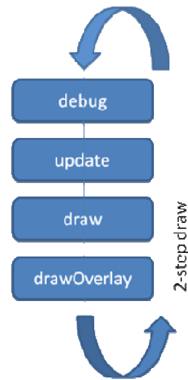
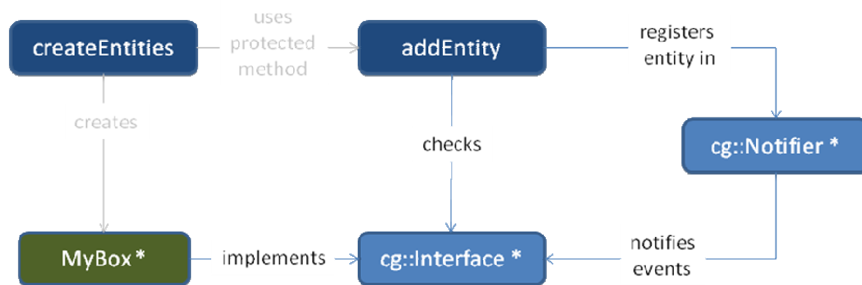


Figure 7 - Execution cycle of the synchronous listeners

For the *box* to appear on the screen *MyBox* must implement *cg::IDrawListener* and its *draw* method.

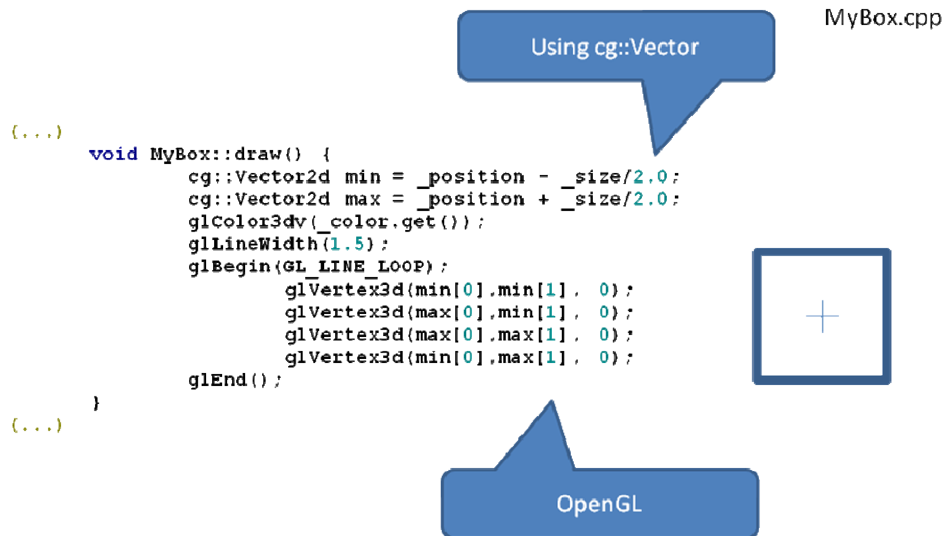


MyBox.h

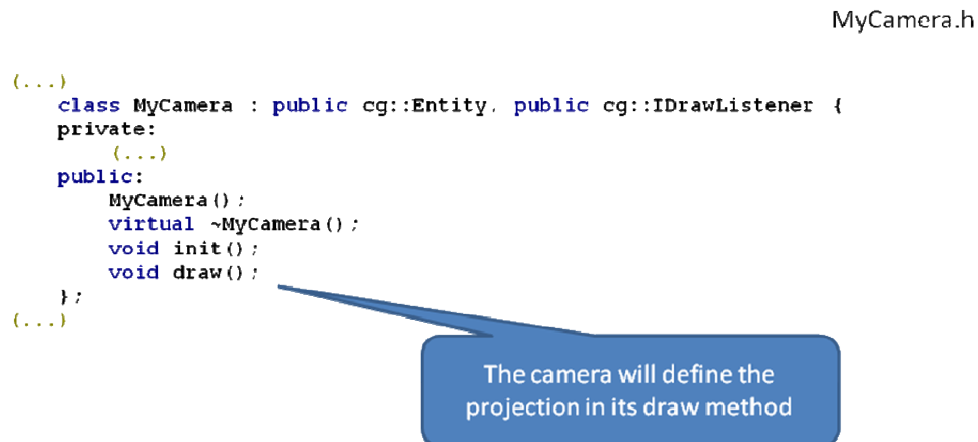
```
(...)
        class MyBox : public cg::Entity, public cg::IDrawListener {
        private:
                (...)

        public:
                (...)
                void draw();
        };
}
(...)
```

implement
cg::IDrawListener

Inside *draw* we will use OpenGL commands to actually draw the *box*. For more information about OpenGL drawing commands, please refer to Chapter 2 of [Shreiner et al. 2005].



Even though an entity is registered in the *IDrawListener* notifier, it can never be seen if there is no camera to define the way this entity is rendered. So, we must create a camera. Create another class, *MyCamera*, which also inherits from *cg::Entity* and implements *cg::IDrawListener* and its *draw* method.



In *draw* we define the projection to be used when rendering the graphical entities of the application. For more information about OpenGL viewing commands, please refer to Chapter 3 of [Shreiner et al. 2005].

MyCamera.cpp

```
(...)
    MyCamera::MyCamera()  : Entity("MyCamera")  {}
    MyCamera::~MyCamera()  {}
    void MyCamera::init()  {
        (...)
    }
    void MyCamera::draw()  {
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glOrtho(0,_winWidth,0,_winHeight,0,-100);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
    }
(...)
```

OpenGL code defining the orthogonal projection

MyApp.h

```
(...)
#include "MyCamera.h"
(...)
```

the camera is always the first cg::Entity

MyApp.cpp

```
(...)
    void MyApp::createEntities()  {
            addEntity(new MyCamera());
            addEntity(new MyBox("Box1"));
    }
(...)
```

<u>Attention</u>: In *MyApp::createEntities* the camera should be the first entity to be added. Otherwise, the entities added before it may not appear on the screen, since OpenGL would not know how to “look at them”.

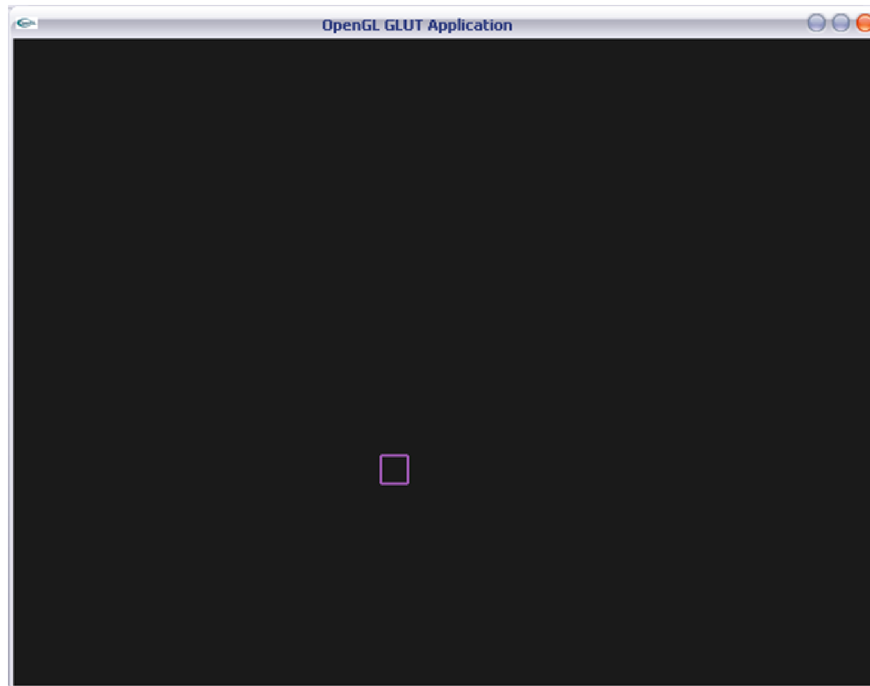


Figure 8 - Result of Step 3

Summary: We drew the *box* on screen by creating a new class, *MyCamera*, which sets up the projection parameter of the graphic pipeline and also by defining *MyBox* as a *cg::IDrawListener*. *MyCamera* inherits from *cg::Entity* and implements *cg::IDrawListener* and defines the projection to be used in the application.

Step 4: Animating the *box*

In this step we will animate *MyBox* with a constant linear velocity and configure it so that it rebounds on screen edges.

To animate an entity, we must implement *cg::IUpdateListener* and its *update* method. This process is very similar to implementing *cg::IDrawListener* and its *draw* method when we want that entity to be drawn on the screen.

MyBox.h

```
(...)
class MyBox : public cg::Entity,
              public cg::IDrawListener,
              public cg::IUpdateListener
{
(...)
public:
        (...)
        void update(unsigned long elapsed_millis);
        (...)
};
(...)
```

implementing the *IUpdateListener* interface

MyBox.cpp

reading from config.ini

creating the box

```
(...)

void MyBox::init() {
    // Read from property file
    double min_size = cg::Properties::instance()->getDouble("MIN_SIZE");
    double max_size = cg::Properties::instance()->getDouble("MAX_SIZE");
    // Creates box
    cg::tWindow win = cg::Manager::instance()->getApp()->getWindow();
    _winWidth = win.width;
    _winHeight = win.height;
    _position = cg::Vector2d(randomBetween(0,_winWidth),randomBetween(0,_winHeight));
    _size = cg::Vector2d(randomBetween(min_size,max_size),randomBetween(min_size,max_size));
    _color = cg::Vector3d(randomBetween(0.1,0.9),randomBetween(0.1,0.9),randomBetween(0.1,0.9));
    _velocity = cg::Vector2d(randomBetween(100,300),randomBetween(100,300));
}
```

time to use in the simulation

```
void MyBox::update(unsigned long elapsed_millis) {
        double elapsed_seconds = elapsed_millis / 1000.0;
        _position += _velocity * elapsed_seconds;
        if(_position[0] < 0) {
                _position[0] = 0;
                _velocity[0] = -_velocity[0]; }
        if(_position[0] > _winWidth) {
                _position[0] = _winWidth;
                _velocity[0] = -_velocity[0]; }
        if(_position[1] < 0) {
                _position[1] = 0;
                _velocity[1] = -_velocity[1]; }
        if(_position[1] > _winHeight) {
                _position[1] = _winHeight;
                _velocity[1] = -_velocity[1]; }
}
(...)
```

uniform velocity

collision with the screen limits (read in *init*)

To make the *box* move, *MyBox* must implement *cg::IUpdateListener* and its *update* method. The *update* method receives the elapsed time since the last update, allowing the simulation to run evenly on any machine, not depending on the machine's processor.

In Figure 9 you can observe the final result of this step: the *box* is moving and it rebounds on the window's edges.

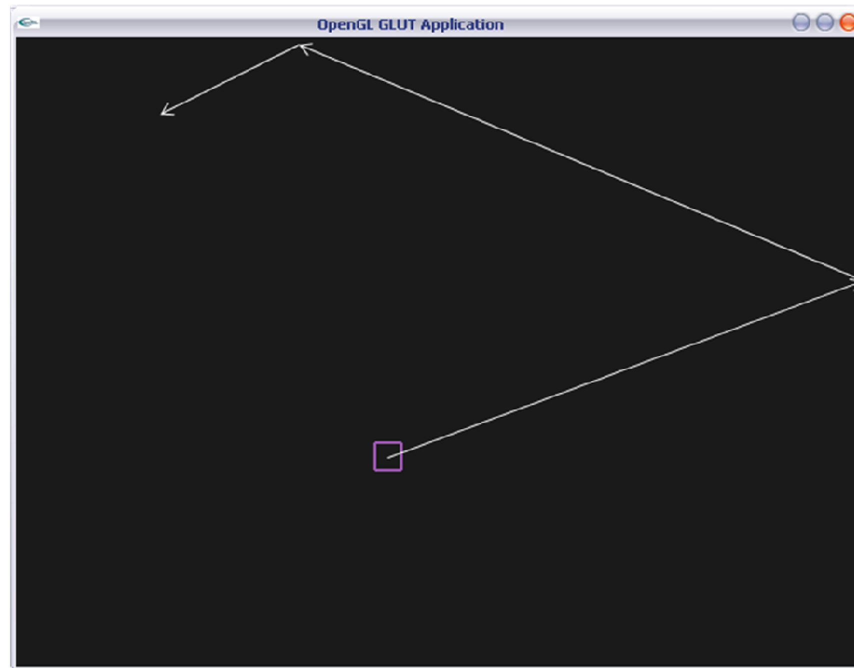


Figure 9 - Result of Step 4

<u>Summary</u>: We now have an animated *box*. To achieve this we defined *MyBox* as a *cg::IUpdateListener*.

Step 5: Create a mouse controlled *bat*

In this step we will create a mouse controlled *bat*.

To create a mouse controlled entity we need to create a class that implements *cg::IMouseEventListener* and its methods *onMouse*, *onMouseMotion* and *onMousePassiveMotion*. These methods are called after events such as a mouse click, a mouse click and drag or a mouse move, correspondingly. The entity's behavior responding to a mouse event should be defined inside these methods.

Create a new class *MyBat* that inherits from *cg::Entity* and implements *cg::IMouseEventListener* and its methods. In the example code the *bat* follows the mouse by setting its position equal to the mouse's position.

MyBat.h

```
class MyBat : public cg::Entity,
              public cg::IDrawListener,
              public cg::IMouseEventListener
{
        private:
                (...)
        public:
                (...)
                void init();
                void draw();
                void onMouse(int button, int state, int x, int y);
                void onMouseMotion(int x, int y);
                void onMousePassiveMotion(int x, int y);
};
```


MyBat.cpp

```
(...)
void MyBat::draw() {
        cg::Vector2d min = _position - _size/2.0;
        cg::Vector2d max = _position + _size/2.0;
        glColor3d(0.9,0.9,0.9);
        glLineWidth(1.5);
        glBegin(GL_LINE_LOOP);
                glVertex3d(min[0],min[1], 0);
                glVertex3d(max[0],min[1], 0);
                glVertex3d(max[0],max[1], 0);
                glVertex3d(min[0],max[1], 0);
        glEnd();
}
void MyBat::onMouse(int button, int state, int x, int y) {}
void MyBat::onMouseMotion(int x, int y) {}
void MyBat::onMousePassiveMotion(int x, int y) {
        _position[0] = x;
        _position[1] = _winHeight - y;
}
(...)
```

MyApp.h

```
(...)
#include "MyBat.h"
(...)
```

MyApp.cpp

```
(...)
	void MyApp::createEntities() {
		addEntity(new MyCamera());
		addEntity(new MyBox("Box1"));
		addEntity(new MyBat("Bat"));
	}
(...)
```

Once again, for the new entity to appear on the screen, *MyBat* must implement *cg::IDrawEventListener* and the *draw* method. Also, we must add this new entity to the application in *MyApp::CreateEntities*.

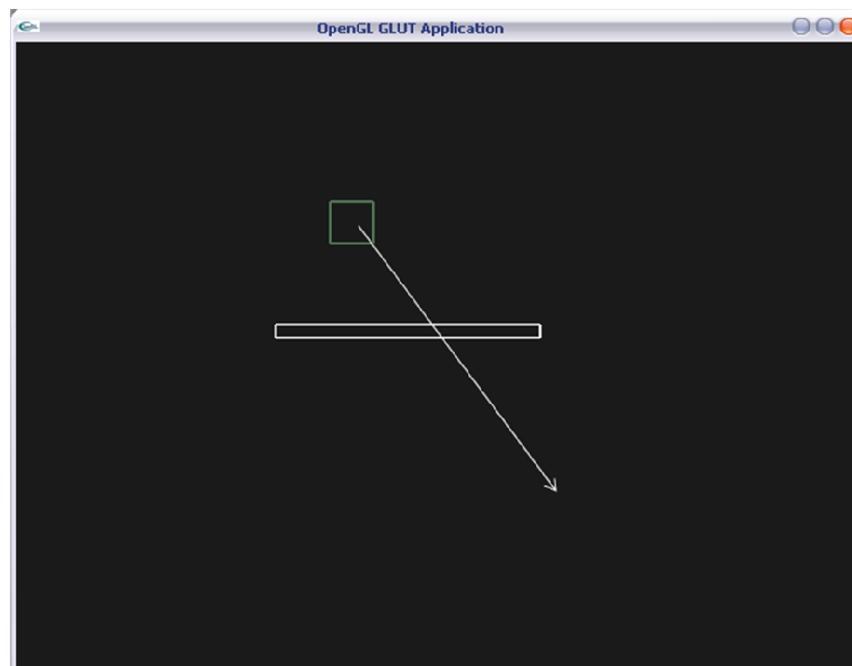


Figure 10 - Result of Step 5

<u>Summary</u>: We created a mouse controlled *bat*. To do this, we created a new *cg::Entity* class, *MyBat*, implementing *cg::IDrawListener* and *cg::IMouseEventListener*.

Step 6: Collision between *bat* and *box*

In this step we will implement collision detection between the *bat* and the *box*es. We will also learn how to get the reference to a registered entity using *cg::Registry*, learn about the singleton *cg::Util* and its collision methods and, finally, implement the collision effect on the animation of *MyBox*.

In order for a *box* to realize it has collided with the *bat*, it needs to have a reference to it. We can get that reference through the *cg::Registry* singleton class. This class keeps a reference to each entity created in the application. When a *cg::Entity* is added to the application through the *cg::Application::addEntity* method, its reference is kept by *cg::Registry*.

MyBox.h

```
(...)
class MyBox : public cg::Entity,
              public cg::IDrawListener,
              public cg::IUpdateListener
{
private:
        (...)
        MyBat* _bat;
public:
        (...)
};
(...)
```

MyBox.cpp

```
(...)
void MyBox::init() {
        (...)
         _bat = (MyBat*)cg::Registry::instance()->get("Bat");
}
void MyBox::update(unsigned long elapsed_millis) {
        (...)
        if(_bat->isCollision(_position, _size)) {
                _velocity[1] = -_velocity[1]; }
}
(...)
```

get reference to the *bat* in cg::Registry

collision effect

MyBat.h

```
(...)
class MyBat : public cg::Entity,
              public cg::IDrawListener,
              public cg::IMouseEventListener {
(...)
public:
        (...)
        bool isCollision(cg::Vector2d box_position, cg::Vector2d box_size);
};
(...)
```

collision detection

use the collision method defined in *cg::Util*

MyBat.cpp

```
(...)
bool MyBat::isCollision(cg::Vector2d box_position, cg::Vector2d box_size) {
        cg::Vector2d bat_bottomleft = _position - _size / 2.0;
        cg::Vector2d bat_topright = _position + _size / 2.0;
        cg::Vector2d box_bottomleft = box_position - box_size / 2.0;
        cg::Vector2d box_topright = box_position + box_size / 2.0;
        return cg::Util::instance()->isAABBoxCollision(
                bat_bottomleft,bat_topright,box_bottomleft,box_topright);
}
(...)
```

To detect the collision we create a method *MyBat::isCollision* that receives a *box*'s position and size. Inside this method we call the already defined method *cg::Util::isAABBoxCollision* (Axis Aligned Bounding Box Collision). This method receives the bottom-left and top-right vertexes of the imaginary box surrounding each object and checks if these boxes are overlapping.

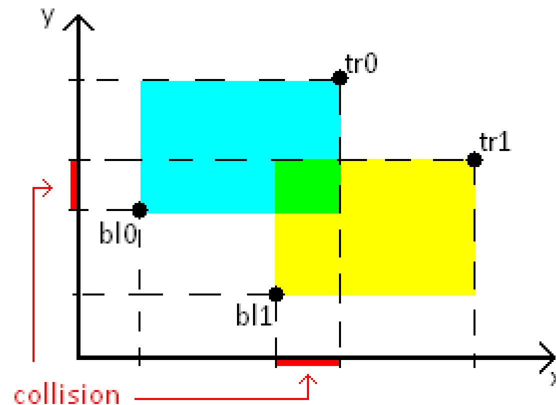


Figure 11 - Axis aligned bounding box collision detection

Creating an imaginary box around the object allows us to detect collisions between irregular objects. In this case, the objects are already box-shaped so this volume is the perfect match.

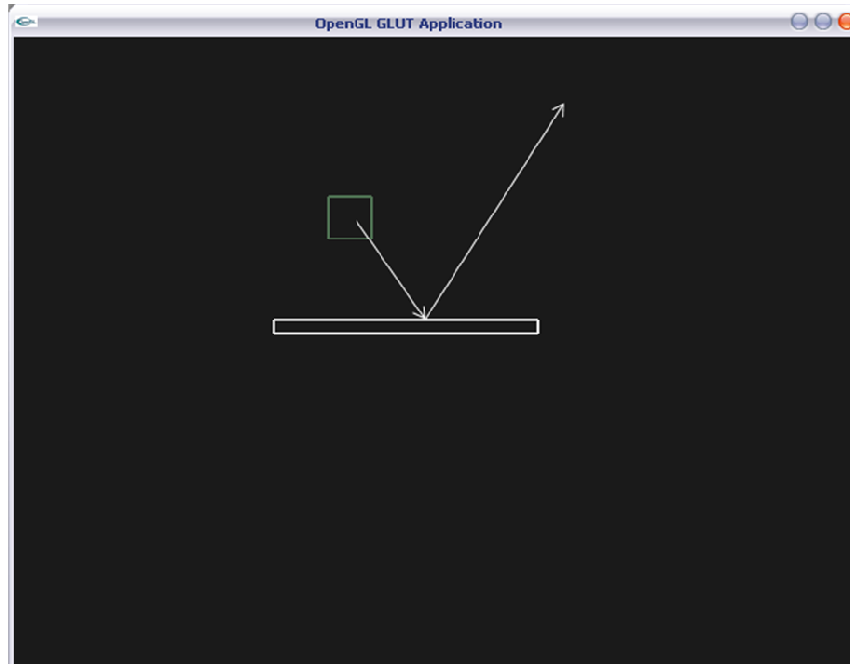


Figure 12 - Result of Step 6

Summary: We are now able to detect collisions between the *box*es and the *bat*. To do this, we added *MyBox* a reference to *MyBat* and each time the *bat* moves we check if it collides with any *box*.

Step 7: Create more *boxes*

In this step we will create more *box*es.

If we want to create more *box*es we need only to call *MyApp::addEntity* inside *MyApp::createEntities* for every *box* we wish to create.

MyApp.cpp

```
(...)
void MyApp::createEntities() {
	addEntity(new MyCamera());
	addEntity(new MyBox("Box1"));
	addEntity(new MyBox("Box2"));
	addEntity(new MyBox("Box3"));
	addEntity(new MyBox("Box4"));
	addEntity(new MyBox("Box5"));
	addEntity(new MyBox("Box6"));
	addEntity(new MyBox("Box7"));
	addEntity(new MyBat("Bat"));
}
(...)
```

not very extensible and inelegant

However, this solution is not very extensible and is also inelegant. During the following step we will see how to solve these problems.

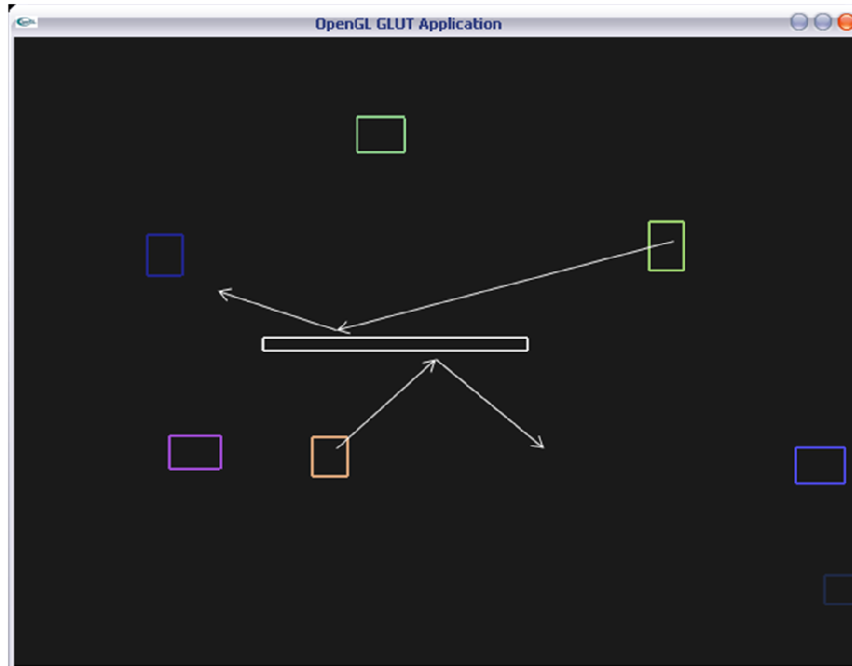


Figure 13 - Result of Step 7

Summary: We now have more *box*es in our application. The classes suffered no changes.

Step 8: Create an arbitrary number of *boxes*

In this step we will see how to create an arbitrary number of *box*es without having to add them, one by one, in *MyApp::createEntities*. Also, we will introduce another *cglib* listener, *IDrawOverlayListener*, which helps us create HUD[4] (Head Up Display) like features.

Let's create an entity manager - *MyBoxManager*. This class will manage all *box*es within *MyApp* and keep them in a *std::vector<MyBox*>*. *MyBoxManager* will also contain a *std::vector<MyBox*>::iterator*, which will allow us to iterate throughout all the *box*es, simplifying the task of handling them.

MyBoxManager.h

```
(...)
class MyBoxManager  : public cg::Entity,
                      public cg::IDrawListener,
                      public cg::IUpdateListener,
                      public cg::IDrawOverlayListener  {
private:
        std::vector<MyBox*>  _boxes;
        typedef std::vector<MyBox*>::iterator  tBoxIterator;
        std::string nboxMessage;

public:
        MyBoxManager(std::string  id);
        ~MyBoxManager();
        void init();
        void update(unsigned long elapsed_millis);
        void draw();
        void drawOverlay();
};
(...)
```

using the STL
(Standard Template Library)

new *IDrawOverlay* interface

How many *box*es do we want the application to have? What happens if we want to change the number of *box*es in the future? As referred earlier, we can, and should, use the configuration file to define application parameters that may be changed. Therefore, let's define an NBOX (number of *box*es) parameter in "config.ini" (ex: NBOX = 77).

Now suppose we want to show on screen how many *box*es have been created and we want it to look like a HUD (i.e., it should be on top of all other entities). Since only the manager knows how many *box*es are being created, the string representing this information should be managed by *MyBoxManager*. However, because the string is static (it does not move nor suffers changes), it makes no sense to create an entity just for displaying the number of *box*es on the screen and for *MyBoxManager* to keep calling its *draw* method. So what we can do is to draw the string inside the *MyBoxManager::drawOverlay* method. *MyBoxManager* can implement this method by implementing the *cg::IDrawOverlayListener* class. The *drawOverlay* listener is the last of the synchronous listeners to be called in the cycle. Therefore, the entities drawn in *drawOverlay* are drawn on top of all the others. Since we just want to draw a string on the screen we can use the *cglib* method *cg::Util::drawBitmapString*.

C++ encourages programmers to be "ecological", so we must destroy all objects that are not being used anymore. In *MyBoxManager*'s destructor you can see that all *box*es are being destroyed along with the manager. This happens because it makes no sense for the *box*es to exist without their manager.

MyBoxManager.cpp

```
(...)
MyBoxManager::~MyBoxManager() {
        for(tBoxIterator i = _boxes.begin(); i != _boxes.end(); i++) {
                delete (*i);
        }
}


void MyBoxManager::init() {
        (...)
        int nbox = cg::Properties::instance()->getInt("NBOX");
        for(int i = 0; i < nbox; i++) {
                std::ostringstream os;
                os << "Box" << i;
                MyBox *box = new MyBox(os.str());
                box->init();
                _boxes.push_back(box);
        }
}
(...)


(...)
void MyBoxManager::update(unsigned long elapsed_millis) {
        for(tBoxIterator i = _boxes.begin(); i != _boxes.end(); i++) {
                (*i)->update(elapsed_millis);
        }
}

void MyBoxManager::draw() {
        for(tBoxIterator i = _boxes.begin(); i != _boxes.end(); i++) {
                (*i)->draw();
        }
}

void MyBoxManager::drawOverlay() {
        glColor3d(0.9,0.1,0.1);
        cg::Util::instance()->drawBitmapString(nboxMessage,10,10);
}
(...)
```


MyBoxManager is responsible for calling the *box*es' methods, so each *box*'s *update* and *draw* methods should be called inside *MyBoxManager*'s own *update* and *draw* methods. Notice that we are using the *std::vector<MyBox*>::iterator* to access each one of the *box*es. For more information on lists and iterators please refer to [Eckel 2000] and [Eckel & Allison 2003].

Due to the fact that the manager is now responsible for handling all the *box*es, they should be created by *MyBoxManager* instead of by *MyApp*. So, we remove all *MyBox* entities creation from *MyApp::createEntities* and place them in *MyBoxManager::init*. The application now needs to have a *MyBoxManager* to replace the *MyBox* entities, so we add one using the *addEntity* method inside *MyApp::createEntities*.

MyApp.h

```
(...)
#include "MyBoxManager.h"
(...)
```

Replace the individual creation of the *boxes*

MyApp.cpp

```
(...)
        void MyApp::createEntities()  {
                addEntity(new MyCamera());
                addEntity(new MyBoxManager("BoxManager"));
                addEntity(new MyBat("Bat"));
        }
(...)
```

As a result, we now have plenty of *box*es appearing on the screen without having to draw them one by one.

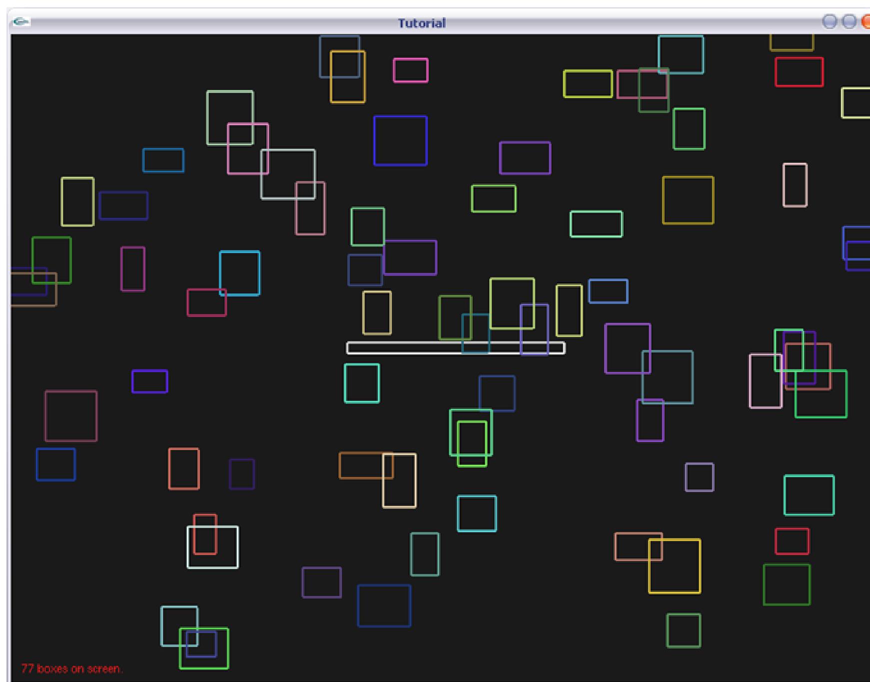


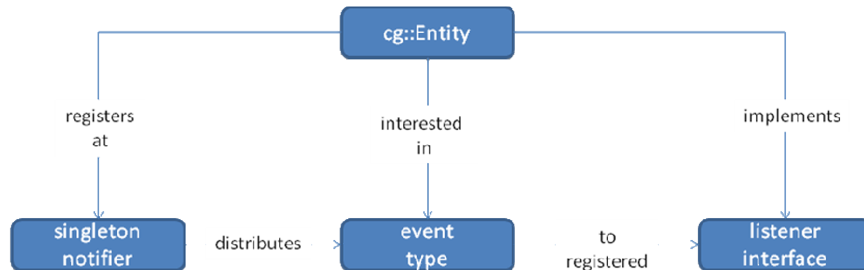
Figure 14 - Result of Step 8

Summary: To control all the *box*es in the application, we created a new class, *MyBoxManager*. For the information on the number of *box*es to appear on screen *MyBoxManager* now also implements *cg::IDrawOverlayListener*.

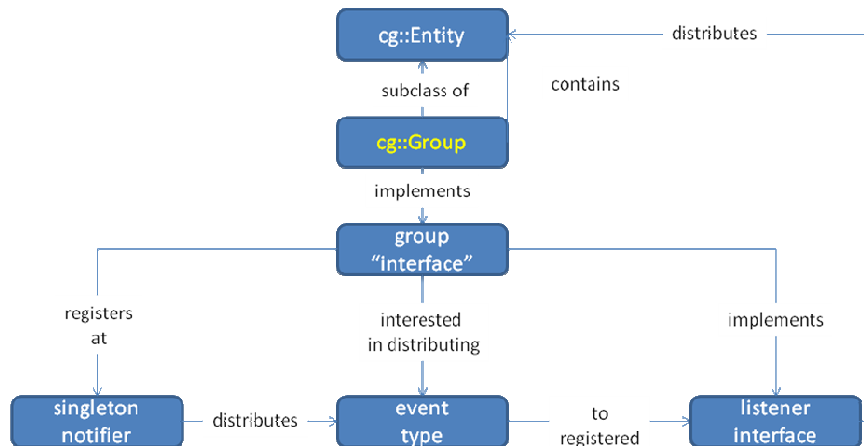
Step 9: Using *cg::Group* to manage entities

In this step we will introduce the *cglib* group classes. We will use *cg::Group* to re-implement the "box manager".

Up till now, even if an entity was being created by a manager, we had to implement each entity's method.



We can simplify this task by using the *cg::Group* class.



A *cg::Group* is an entity that contains other entities. It can implement the same interfaces as any entity plus the specific group classes. These group classes automatically distribute events to the inner entities.

To do this, we modify *MyBoxManager* so that it inherits *cg::Group* instead of *cg::Entity*, this way giving it immediate control on all contained entities. *MyBoxManager* will now also implement *cg::GroupDraw* and *cg::GroupUpdate* instead of *cg::IDrawListener* and *cg::IUpdateListener*. The only class implementation that will remain unchanged is *cg::IDrawOverlayListener*, since it deals only with the number of *box*es. We do not use *cg::GroupDrawOverlay* instead of *cg::IDrawOverlayListener* because we

want the number of *box*es to be drawn only once and not by every group entity. By implementing *cg::GroupDraw* we are implementing *cg::IDrawListener* and automatically distributing the *draw* event to all inner entities. The same happens with *cg::IUpdateListener*.

MyBoxManager.h

```
(...)
class MyBoxManager  : public cg::Group,
                      public cg::GroupDraw,
                      public cg::GroupUpdate,
                      public cg::IDrawOverlayListener
{
(...)
protected:
        void createEntities();
        void postInit();
public:
        MyBoxManager(std::string  id);
        ~MyBoxManager();
        void drawOverlay();
};
(...)
```

creation of *cg::Group*'s entities

method executed after the *cg::Entity's init method*

The creation of entities in a *cg::Group* must be done in the *createEntities* method. So, *MyBoxManager* now implements *cg::Group::createEntities*. It also implements the *cg::Group::postInit* method to create the message we want to be written in the *drawOverlay* method.

MyBoxManager.cpp

```
void MyBoxManager::createEntities()  {
        int nbox = cg::Properties::instance()->getInt("NBOX");
        for(int i = 0; i < nbox; i++) {
                std::ostringstream os;
                os << "Box" << i;
                add(new MyBox(os.str()));
        }
}
void MyBoxManager::postInit()  {
         std::ostringstream os;
         os << size() << " boxes on screen.";
         _nboxMessage = os.str();
}
void MyBoxManager::drawOverlay()  {
         glColor3d(0.9,0.1,0.1);
         cg::Util::instance()->drawBitmapString(_nboxMessage,10,10);
}
```

protected method in *cg::Group*

If we run the application, the result appears to be the same. However, in the application code, the way the *box*es are being drawn is much different.

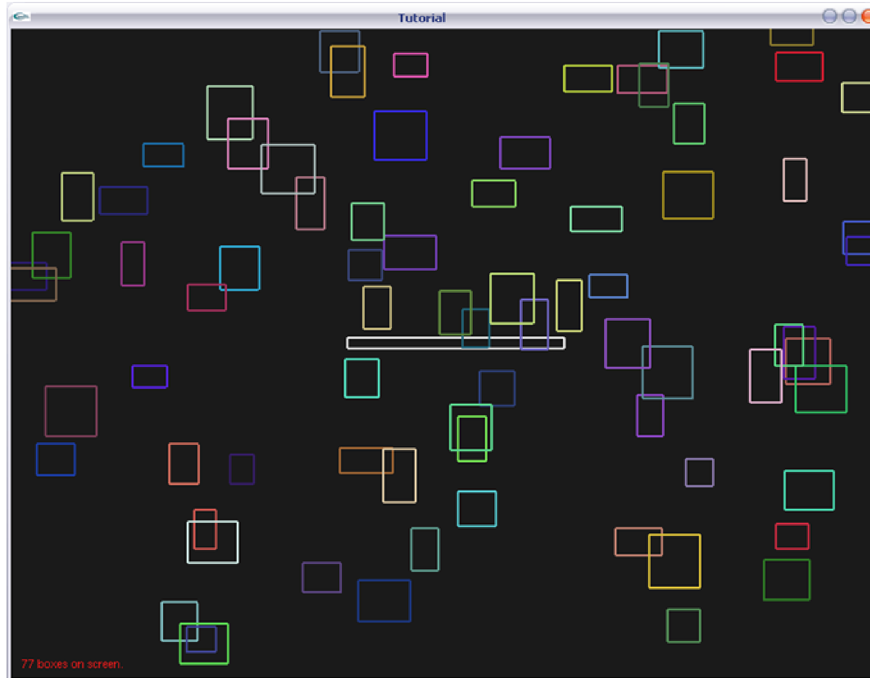


Figure 15 - Result of Step 9

Summary: To take advantage of the *cg::Group* capabilities, *MyBoxManager* now inherits from *cg::Group* and implements the *cg::GroupDraw*, *cg::GroupUpdate* and *cg::IDrawOverlay* interface classes.

Step 10: Application shutdown and memory leaks

In this step we will learn how to debug for memory leaks using VS2005. We will also learn how to handle keyboard events.

Memory Leaks

Memory leaks occur due to the fact that memory slots, which were occupied by objects that are no longer being used, have not been released. VS 2005 allows us, during runtime, to check which memory slots were not cleaned up. To use this capability, we must add the following code in "main.cpp":

main.cpp

```
//BEGIN: MEMORY LEAK DETECTION
#define _CRTDBG_MAP_ALLOC
#include <stdlib.h>
#include <crtdbg.h>
//END: MEMORY LEAK DETECTION

(... includes ...)

int main(int argc, char** argv) {

//BEGIN: MEMORY LEAK DETECTION
        _CrtSetDbgFlag ( _CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF );
        //_CrtSetBreakAlloc(xxx);
//END: MEMORY LEAK DETECTION

        (... Codigo ...)
}
```


This set of instructions allows VS 2005 to detect memory leaks and write them in the output window.

Here is an example of a detected memory leak:

(output)

```
Detected memory leaks!
Dumping objects ->
c:\program files\microsoft visual studio 8\vc\include\crtdbg.h(1147) : {1364} normal block at
0x08AEEBA0, 140 bytes long.
 Data: <$ P  bQ  XQ H    > 24 0A 50 00 80 62 51 10 0C 58 51 10 48 8A AE 08
c:\program files\microsoft visual studio 8\vc\include\crtdbg.h(1147) : {1363} normal block at
0x08AEEB30, 52 bytes long.
 Data: <0   0   0        > 30 EB AE 08 30 EB AE 08 30 EB AE 08 CD CD CD CD
```

In this case, 1363 represents the 1363rd object allocated since the application started. If we want to know when the memory was allocated we need to give the highlighted number as an argument for the function `_CrtSetBreakAlloc(long _BreakAlloc)` and run the application in debug mode.

main.cpp

```
int main(int argc, char** argv) {

//BEGIN: MEMORY LEAK DETECTION
        _CrtSetDbgFlag ( _CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF );
        _CrtSetBreakAlloc(1363);
//END: MEMORY LEAK DETECTION

        (... Codigo ...)
}
```

Application Shutdown and Keyboard Events

We now want to make the application shut down cleanly by pressing the “Esc” key. Additionally, we want the *box*es to appear/disappear when we press the “space” key and the application structure to be dumped to “log.txt” when we press “F1”.

Running this tutorial’s application creates dumps of allocated memory that has not been freed. This happens due to the fact that none of the created entities are being deleted.

We will create a new class *MyController* inheriting from *cg::Entity* and implementing the *cg::IKeyboardEventListener* so it can detect the “Esc”, “space” and “F1” keys. We will implement the *onKeyPressed*, *onKeyReleased* and *onSpecialKeyReleased* methods to handle the events of pressing “Esc”, “space” and “F1”, correspondingly. Special keys are the arrow keys and the “F” keys. To finish, we will add the new controller entity to *MyApp*.

MyController.h

```
(...)
class MyController  : public cg::Entity,
                      public cg::IKeyboardEventListener
{
public:
        MyController();
        ~MyController();
        void init();
        void onKeyPressed(unsigned char key);
        void onKeyReleased(unsigned char key);
        void onSpecialKeyPressed(int  key);
        void onSpecialKeyReleased(int  key);
};
(...)
```

IKeyboardEventListener

MyController.cpp

clean shutdown

```
void MyController::onKeyPressed(unsigned char key) {
        if (key == 27) {
                cg::Manager::instance()->shutdownApp();
        }
}
void MyController::onKeyReleased(unsigned char key) {
        if (key == ' ') {
                cg::Registry::instance()->get("BoxManager")->state.toggle();
        }
}
void MyController::onSpecialKeyPressed(int  key) {
}
void MyController::onSpecialKeyReleased(int  key) {
        if (key == GLUT_KEY_F1) {
                        cg::Manager::instance()->getApp()->dump();

}
```

changes the *BoxManager’s* state

GLUT special keys (arrow keys and F1 to F12)

structure dump to log.txt

MyApp.h

```
(...)
#include "MyController.h"
(...)
```

MyApp.cpp

```
(...)
    void MyApp::createEntities()  {
        addEntity(new MyCamera());
        addEntity(new MyBoxManager("BoxManager"));
        addEntity(new MyBat("Bat"));
        addEntity(new MyController());
    }
(...)
```

As discussed earlier, we should free all allocated memory when the application shuts down. The *cg::Manager::shutDownApp* method already does that work for us, by cleaning each one of the notifiers and deleting the application itself. This way we will have no memory leaks in *MyApp*.

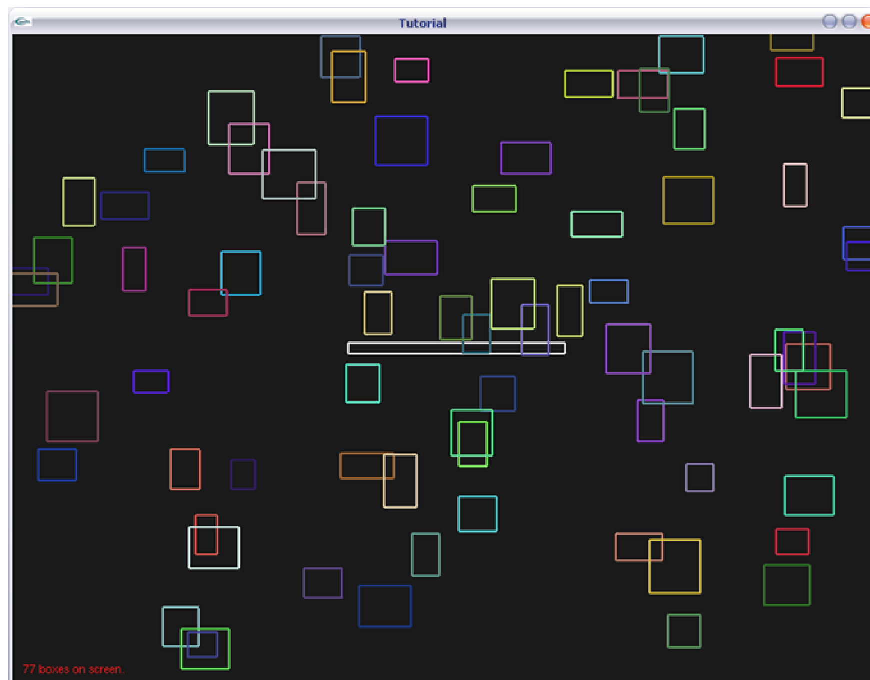


Figure 16 - Result of Step 10

<u>Summary</u>: To shut down the application cleanly, we created a new *cg::Entity* class named *MyController* that implements the *cg::IKeyboadEventListener*, allowing it to handle keyboard events and handle application shutdown adequately.

Step 11: Adapting to window resizing

In this step we will learn how to handle window resizing.

If we try running the application and changing the window's size the *box*es appearance will be modified (see Figure 17). Also, the *bat* continues following the mouse but in a different position.

Figure 17 - *MyApp*'s window resized

MyCamera.h

```
(...)
class MyCamera  : public cg::Entity,
                  public cg::IDrawListener,
                  public cg::IReshapeEventListener  {
(...)
public:
        (...)
        void onReshape(int width, int height);
};
(...)
```

MyCamera.cpp

```
(...)
void MyCamera::draw()  {
        (...)
        glOrtho(0,_winWidth,0,_winHeight,0,-100);
        (...)
}
void MyCamera::onReshape(int width, int height)  {
        _winWidth = width;
        _winHeight = height;
}
(...)
```

To solve these problems we will use *cglib*'s *cg::IReshapeEventListener*. Implementing this listener will allow classes to be able to respond to window reshape events without problems such as above.

The first class we need to change is *MyCamera*. Entities are being deformed because the window is being reshaped and the camera does not know that. So the camera keeps looking at the entities as if they were inside the old window, therefore deforming their appearance. We must adjust the projection to the new window and this is done by giving *MyCamera* the new window size.

Yet, we still have other problems to solve. Figure 18 shows the application window which is 640 pixels wide and 480 pixels high. However, notice that the *bat*'s coordinates are different depending on weather we are considering window coordinates (x, y) = (143, 9) or OpenGL coordinates (x, y) = (143, 471).

Figure 18 – Window coordinates and OpenGL coordinates

This happens due to the fact that window coordinates and OpenGL coordinates have a different yy axis orientation. When reshaping the window, we must change the *bat*'s yy position accordingly, otherwise the *bat* will appear dislocated from the mouse's position. To do this, *MyBat* must also implement *cg::IReshapeEventListener*.

MyBat.h

```
(...)
class MyBat : public cg::Entity,
              public cg::IDrawListener,
              public cg::IMouseEventListener,
              public cg::IReshapeEventListener
{
(...)
public:
        (...)
        void onReshape(int width, int height);
};
(...)
```

MyBat.cpp

```
(...)
void MyBat::onMousePassiveMotion(int x, int y) {
        _position[0] = x;
        _position[1] = _winHeight - y;
}
void MyBat::onReshape(int width, int height) {
                _winHeight = height;
        }
(...)
```

We must also recalculate the *box*es' collision limits. This problem is solved if *MyBox* also implements *cg::IReshapeEventListener*.

MyBox.h

```
(...)
class MyBox : public cg::Entity, public cg::IDrawListener,
              public cg::IUpdateListener,
              public cg::IReshapeEventListener   {
(...)
public:
        (...)
        void onReshape(int width, int height);
};
(...)
```

MyBox.cpp

```
(...)
void MyBox::update(unsigned long elapsed_millis) {
        (...)
        if(_position[0] > _winWidth) { (... collide ...) }
        (...)
}
void MyBox::onReshape(int width, int height) {
        _winWidth = width;
        _winHeight = height;
}
(...)
```

Finally, one last remark: if we need to define an initial default window size, we may do so in *MyApp*'s constructor.

MyApp.cpp

```
(...)
MyApp::MyApp() : cg::Application("config.ini") {
        _window.caption = "Tutorial";
        _window.width = 800;
        _window.height = 600;
}
(...)
```

As you can see in Figure 19, all the problems with the *box*es' shapes, the *bat*'s location and the collision limits, are now solved.

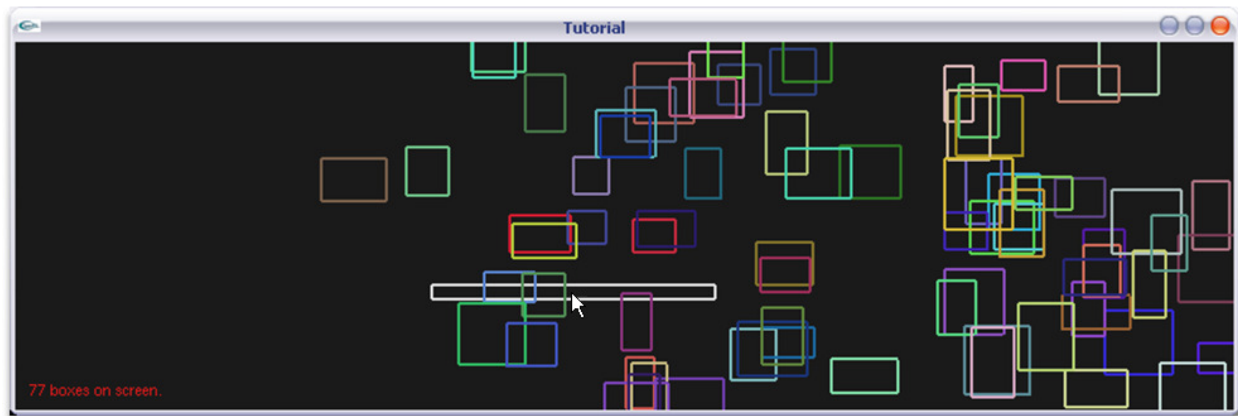


Figure 19 - Result of Step 11

Summary: For the *box*es appearance and the *bat*'s position not to be modified when the window is resized, *MyBat*, *MyBox* and *MyCamera* now implement *cg::IReshapeEventListener* and incorporate the new size information in their computation.

Glossary

[1]***Cglib*** – Object oriented library which works as a layer between an application and OpenGL and GLUT. It helps to structure and simplify code production for applications with 2D/3D graphics.

[6]**Entity** – Subclass of *cg::Entity* that represents an object with its own behavior. This object may be drawn, updated and debugged. It is a base element of the application.

[5]**Frame Rate** – Number of frames per second (fps), i.e., number of times the application updates the scene in a second. This parameter has an ideal value of 60 fps. In a slower machine it can be decreased down to 30 fps. Below the value of 10 fps the animation will look segmented, therefore loosing quality.

[3]**GLUT** (OpenGL Utility Toolkit) – Library of utilities for OpenGL programs, which primarily perform system-level I/O with the host operating system. Functions performed include window definition, window control, and monitoring of keyboard and mouse input.

[4]**HUD** (Head-Up Display) - Method by which information is visually relayed to the player in computer and video games. Many video games use HUDs to show information on top of the main image.

[2]**OpenGL** (Open Graphics Library) – Standard specification defining a cross-language cross-platform API for writing applications that produce 2D/3D computer graphics.

[7]**Singleton** - Class that is supposed to have only one instance at any time, therefore it is not meant to be freely instantiated.

References

[Eckel 2000] Eckel, B. (2000): *Thinking in C++ - Volume One: Introduction to Standard C++.*

[Eckel & Allison 2003] Eckel, B., & Allison, C. (2003): *Thinking in C++ - Volume 2: Practical Programming.*

[Shreiner et al. 2005] Shreiner, D., Woo, M., Neider, J., & Davis, T. *OpenGL Programming Guide.*