# Astro Crisis!

**An OpenGL project for Comp315-Advanced programming course**

# PROJECT REPORT

UNIVERSITY OF KWAZULU-NATAL
INYUVESI YAKWAZULU-NATALI

School of Computer Science

May, 2015

| Author | Student Number | Signature |
|---|---|---|
| Liam Barnard | 212508966 | |
| Kreason Naidoo | 212503474 | |
| Shaherin Dehaloo | 212504675 | |
| Muhammad Bassa | 211517805 | |

## Acknowledgements

We would like to thank our lecturer, Dr. D Moodley, for your guidance during this project. Without your lectures on the language

*To all of our friends and families who stood by us through this s*

## Abstract

The overall goal of the project aims to develop a game in C++ using openGL libraries for graphical implementations. The game should exhibit the techniques and styles learned during the course of the module, such as inheritance and polymorphism, as well as class-oriented structures and multiple program applications. The project aims to introduce the course members to the C++ programming language in such a way that they become familiar with its concepts, its uses, and its shortcomings – for example, memory management is an incredibly powerful concept, however it can also lead to memory leaks that can cause a program to effectively slow down an entire computer system.

The project covered in this document is called "Astro crisis" (in honour of the 1979 game "Asteroids" developed by ATARI) – a 3D, space-based defense shooter. The objective of the game is to defeat  wave after wave of asteroids and prevent them from colliding with the planet. Should the planet take enough damage, it is destroyed and the game is considered a loss – the player must either play again or may exit the application.

This project's aim is to achieve features such as

- collision detection
- multiple classes
- lighting effects such as ambient light, ray casting, muzzle flash
- basic keyboard and mouse controls
- graphically represented objects
- textures and model imports
- asteroid split on death
- sounds
- multiplayer
- explosion effects
- Level progression
- HUD

by using all available course material, as well as additional sources to aid in the design and implementation of the required features. GitHub has been used extensively as a central repository for code throughout the project lifespan, and Blender has been used to produce the model used to represent the player

A majority of the features listed above have been achieved, producing a final product that operates smoothly under normal circumstances, and adheres to all of the programming techniques learned throughout the semester, in addition to a few extras gleaned along the way.

## Table of Contents

## 1.0 - Table of Figures

## 2.0 - Introduction

As stated previously, this project aimed to implement a 3-dimensional game using OpenGL libraries for the C++ programming language. The problem area of this project involved developing an ambitious and complicated software solution in a language that most of the team members were unfamiliar with, as well as engaging a whole new facet of computer programming – computer graphics.The environment used during the coding process of the project was the CodeBlocks v13.12 IDE, utilizing the GNU GCC Compiler to generate the machine executable code.

There were several concepts that required deep engagement with the code on a level that many of the team members were previously unaware of. Due to the nature of C++, and its close association with the system memory, the programmer is required to keep track of each variable assignation and ensure that no memory leaks occur. In the event that a memory leak does occur, it becomes very apparent, as the program gradually begins to lag, and eventually the system slows to a point where either it forcefully terminates the program, or needs to be restarted.

In addition to the above, the initial progress of learning the available libraries of OpenGL and their multitudinous uses proved to be a challenge in of itself. The idle animation render functions  ** EDIT

Points to speak on:

- C++ memory management
- pointers and addressing
- openGL basics – shapes and display
- texturing
- changing of perspective
- object loading
- animation
- additional effects such as lighting

end problem definition and problem area

Project objectives:

- create a multi-platform 3D game using the above
- utilize coding techniques such as inheritance, classes, structs, polymorphism
- implement graphical elements such as model imports, lighting and texturing effects

end project objectives

Justification of project:

- learn the coding language of C++
- move away from the safety of a familiar language such as java, and become multilingual programmers (benefit)
- Create an entertaining means of achieving the above objectives

end justification

## 3.0 - Preoject Analysis, Design, and Implementation

### Design and Analysis phase

The project is a space-based asteroid shooter, where the player takes the role of a spaceship attempting to fend off waves of asteroids before they collide with the "Home Planet", or the objective. This particular topic was chosen for two reasons; as a tribute to the 1979 game "Asteroids" developed by Atari for multiple platforms; and  due to the wide range of potential expansion (at the time, "scope creep" was absent in group discussions) that this topic allows for. An iterative approach was adopted for the duration of the project

The initial analysis phase saw the breakdown of the project specifications into a set of "target areas" around which development would focus. These target areas focussed on class design and attribute considerations - however, room was left for implementing areas that were not thoroughly understood or else required a potential redesign. The shift into the design phase saw the completion of a basic UML diagram *(Fig 3.1)* that mapped out the

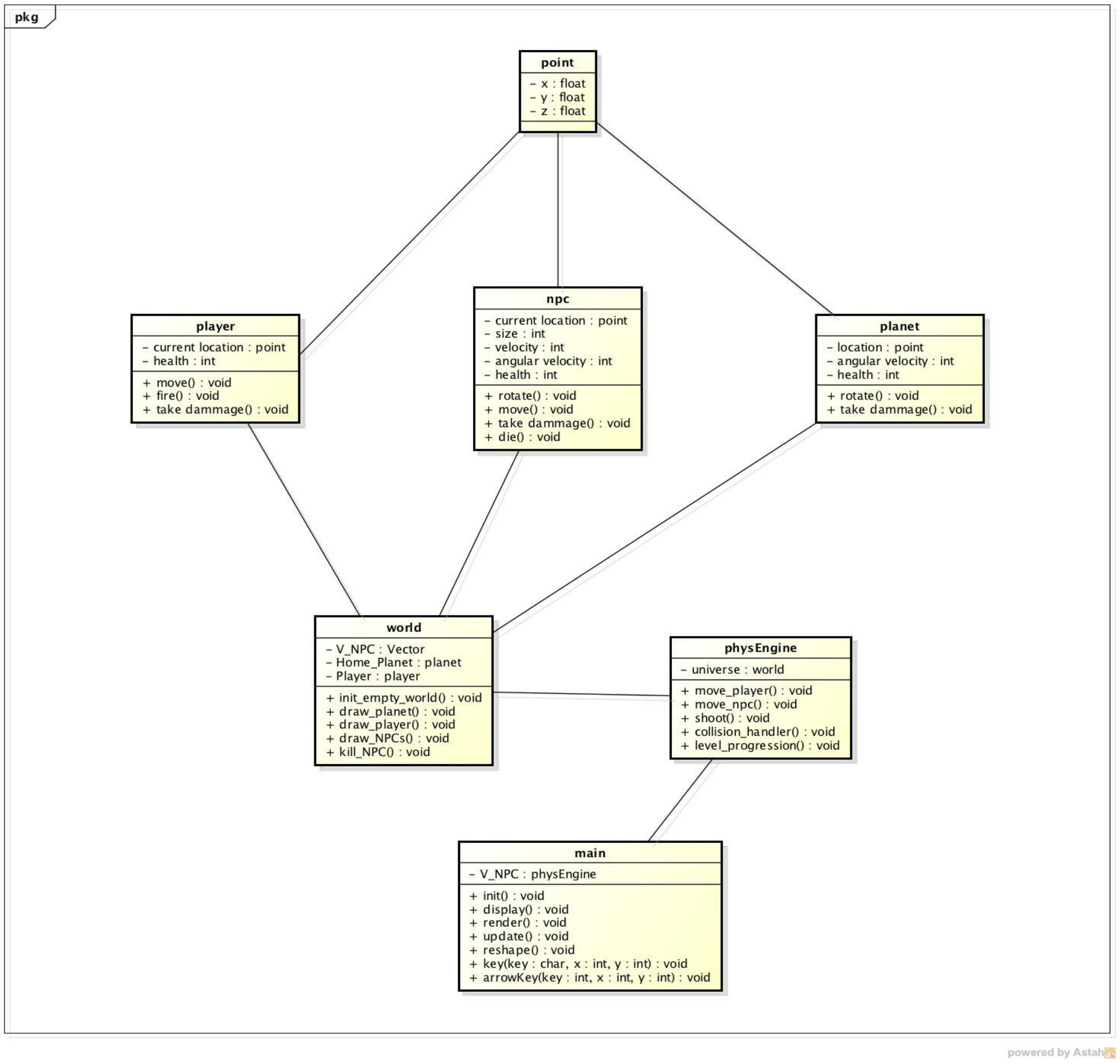


*Fig. 3.1 UML Diagram*

Whilst Fig 3.1 contains most of the class information used at the beginning of the project, the development iterations and expansion of the scope of the project design means that it does not reflect the end system. Rather, it documents the early stages of the project before additional functionality and further graphics were included. The final UML that does accurately reflect the system is Fig 3.2, discussed later on.

The "point" class represented in Fig 3.1 was initially considered as a struct, since it lacks methods. However, upon consideration of the potential expansions to the project, the decision was made to create a point class in case of future expansions such as a distance method, or centerpoint-based collision detection. Every successive class for each renderable object is given an attribute of type 'point' to map out their x,y,z coordinate in the game world, allowing for full 3-dimensional movement (or placement) of objects. The player, planet, and NPC classes were then given further attributes to express unique characteristics, such as angular velocity for the planet and npc classes. As the planet object is static in terms of the game world, move functionality was omitted.

The physics_engine and world classes defined a skeletal game engine, having very little in terms of attributes, but instead comprising mostly of methods – thus closely linking them to the UML definition of an interface (*ch14, Object-Oriented Systems Analysis and Design Using UML, 4th edition*). However, these classes featured heavily in the later implementation phase as the requirements of the various implementations of functionality were understood.

The second phase of development started shortly after the completion of the first milestone, and comprised mostly of class redesign. Multiple instances of memory leaks caused the program to lag severely, despite its limited functions and incredibly basic graphics implementations. To this end, it was proposed that the vector used to store instances of the npc class be altered to instead store pointers to npc objects, as well as the member accessor (->) being used to assign values to the x,y,z coordinates rather than using seperate variables that could potentially add to the overall memory used by the program. Up until this point, there had been very little consideration as to how collision detection could be designed. After reviewing several different documents (simple AABB collision, en.wikipedia.org/wiki/Collision_detection)as to how this could be achieved, it was decided to implement 3 cases using the following algorithms:

```
//Collisions between asteroids
for (number of asteroids in the vector)
        pick an asteroid, i
        for every asteroid j not equal to i
                test the distance between them
                if (distance > product of i and j radii)
                        set collision of asteroid i = true
                        set collision of asteroid j = true
                        assign damage to i and j
                end if
        end for
end for


//Collisions between asteroid and planet
```

```
for (number of asteroids in the vector)
        pick an asteroid, i
                test the difference between the asteroids location and the planets location
                if (distance > product of asteroid i and planet radii)
                set collision of asteroid i = true
                set collision of planet = true
                assign damage to i and j
        end if
end for

//Collisions between the bullet and asteroids
```
***this will be omitted as it follows the same logic as the above**

Once the collision detection had been designed, and any potential memory leaks worked around, further implementation was performed to integrate the designs into the code. At this point, the project began to represent a rough version of the intended final product. Visual and material effects had been added in to the objects after the previous milestone. Player controls had been roughly implemented, but were lacking smooth animation. Mouse-camera control design was originally based off of an online tutorial (http://www.morrowland.com/apron/tutorials/gl/gl_camera_3.zip), however this was eventually scrapped due to a simpler solution being proposed, based off a camera class being designed to handle the mouse movement and translate that into the openGL lookAt() function.

The third and final phase of design centered around the design of the proposed additional features from the milestone 3 document, namely level progression, further movement smoothing, shooting limiter, ray casting, player Heads-Up Display, sounds, model imports, and a credits screen. As of the end of the project, a majority of these have been developed.

The final UML, updated with all the relevant information documenting the classes, their methods and attributes can be found in the resource folder.

## Implementation phase

The classes implemented in the first-phase implementation matched the UML diagram to produce the rough outlines of the code, which was expanded over the course of the project. The end result produced multiple classes that almost exclusively act through the physics_engine class, which serves as the game engine of our program.

Below, each of the classes is discussed in detail, in an attempt to clarify any method implementations as well as cite any references to external source code used.

### Bullet

The bullet class is designed to keep track of the player-asteroid interactions. The bullet class is linked to the mouse click action, so that each time the player clicks, a new bullet object is created at a given x, y, z coordinate that corresponds to the front of the player model. The bullet is given default properties such as its velocity and a set size property. The bullet path is determined by a combination of information – first off, the starting location is given by the players current location. Then the final destination of the projectile is determined by projecting a ray through the x, y mouse coordinates and determining the "contact point" with the outer world sphere. This will be discussed more in the player class description.

The Bullet class features standard draw-to-screen and animation functions such as update() and render(), as well as destructors to remove instances from memory once they go out of scope.

### Cam

In order to achieve the mouse-camera control functionality, it was required to have some means of harvesting the mouse's on-screen coordinates and translate these to the GLUT lookAt() function. Cam achieves this by storing the parameters for the lookAt function, allowing for easy access and manipulation of the information in main. As the asteroids only approach from a single direction, it becomes irrelevant for the player to have full rotational capabilities. To this end, only a half-sphere of view movement has been permitted. Shown below is the code snippet that allows the mouse movement to alter viewing direction:

```
kam -> lx = 0.1*(glutGet(GLUT_WINDOW_WIDTH)/2 - x);
      //^this value denotes the speed of the camera rotation in the x-direction
kam -> ly = 0.1*(glutGet(GLUT_WINDOW_HEIGHT)/2 - y);
      //^this value denotes the speed of the camera rotation in the y-direction
```

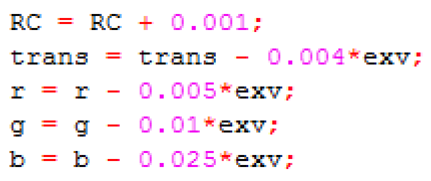
*Fig 3.3*

## Explosions

In order to handle the event of an asteroid being destroyed by either a collision with a bullet, or with the planet, the explosions class developed to supply an animation to announce this event. This class is transient in nature, each instance being created on the death of an asteroid object and only remaining for a brief period thereafter. Note, asteroids that do not collide do not produce explosion objects, reducing the amount of overall memory used.

This class is created once an instance of an asteroid has its *alive* attribute changed to false, and is removed from the vector and thus from the game. During its brief existence,  the explosion object will be subject to several GLUT material display functions, giving it material properties, and blending these functions across its surface. Each explosion has a limited lifespan, as denoted by its RC and RF attributes. As the objects existence extends, the colour is faded from yellow, to red, and finally to black before the object is deleted.

The code block (fig 3.3) alongside shows the functional code that causes the colour change. The variables r, g, and b each correspond to a particular colour used in the rendering of the sphere at any given point.



```
RC = RC + 0.001;
trans = trans - 0.004*exv;
r = r - 0.005*exv;
g = g - 0.01*exv;
b = b - 0.025*exv;
```

*Fig 3.4*

## Imageloader

This class has been adapted from code found on www.codeincodeblocks.com/2012/05/simple-method-for-texture-mapping-on.html. It converts an image into a format readable by the program, and then saves the image ID as a variable of type GLuint. This is then usable by other classes to load the textures onto the object and map to the surface as appropriate

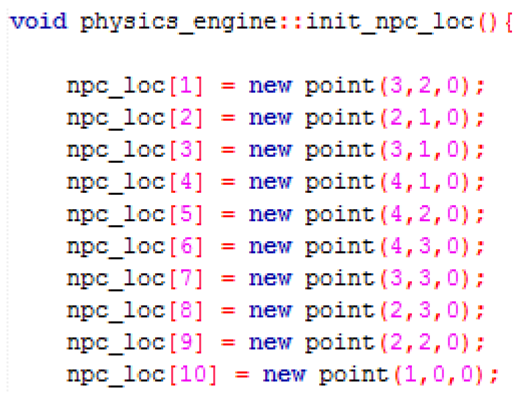
## Model_obj

As with the imageloader, this code was adapted from a web source. In order to import models created by other programs, it is necessary to read in the values of each vertex coordinate on the .obj file.
This is achieved by reading the .obj file line by line, collecting each 3D set of coordinates that maps a single vertex, as well as finding the faces that exist between the groups of vertices as a triangular plane. This is then rendered in the game environment as a single object.
The code can be found at https://tutorialsplay.com/opengl/2014/09/17/lesson-9-loading-wavefront-obj-3d-models

## NPC

One of the biggest classes in this project, the npc class handles the hostile npc's that the player aims to defeat.

The lifespan of an asteroid object begins in the physics_engine class, where it is instantiated using a set of predetermined spawn locations to mark off where each asteroid starts. Each asteroid has a location along a set of points near the world origin that acts as its "go to" point, which is randomly selected. Asteroids come in 3 size categories, being either small, medium, or large; This is reflected by their size attribute, which simply increases the size of their glutSolidSphere. The asteroids are then spawned in, and sent along their paths at a particular velocity.



```
void physics_engine::init_npc_loc(){

    npc_loc[1] = new point(3,2,0);
    npc_loc[2] = new point(2,1,0);
    npc_loc[3] = new point(3,1,0);
    npc_loc[4] = new point(4,1,0);
    npc_loc[5] = new point(4,2,0);
    npc_loc[6] = new point(4,3,0);
    npc_loc[7] = new point(3,3,0);
    npc_loc[8] = new point(2,3,0);
    npc_loc[9] = new point(2,2,0);
    npc_loc[10] = new point(1,0,0);
```

*Fig 3.5 - Code snippet depicting a fragment of the npc spawn location method*

Throughout the duration of each level, the collision detection algorithm monitors the relative position of each asteroid to the planet and bullet objects. Should any enter into the vicinity of another, the collision state of the colliding objects will be set to true and damage is applied to the relevant participants.
In the event that an npc's health attribute falls to a value of 0, it is considered dead and  is deleted from the vector with the die() function.

Any asteroids that go past the planet exist only for a brief time before they are deleted, in order to minimise memory use.

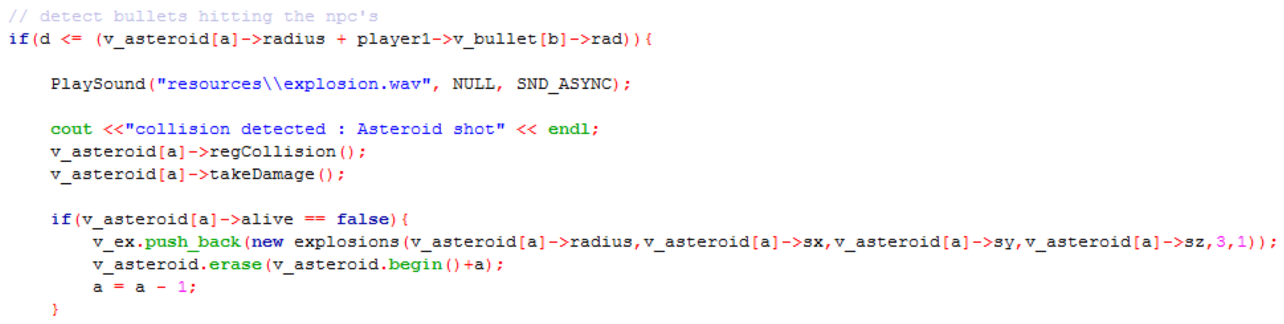
## Physics_engine

This is the primary class of the application. Its purpose is to act as a central class through which all other classes operate to create the gaming environment – a minimal game engine. To this end, all of the objects are declared inside the physics_engine class in one of the multiple pointer vector structures that form its attributes. NPCs and explosions are tracked through the physics_engine, taking into account when their lifespan ends, upon which they are removed from the vector. The justification  behind the use of a vector is due to its dynamic nature – the number of elements contained within is constantly changing. As the planet, world, and player classes contain only a single instance, their instantiations are simply pointers.

To further the catering nature of this class, the asteroids that are created and stored in the vector are also textured for the duration that they are rendered. This prevents a repetitive memory leak that drains massive amounts of RAM, where the image is rendered over and over again on top of itself, eventually resulting in all available memory being used up and the application closing.

All animation functions are dealt with inside the physics_engine class using the update_with_time() function. Each updates with time, resulting in the necessary changes in placement or rotation that give the effect of movement.

One of the most important functions of the physics_engine class is to detect interactions between objects – as mentioned in the npc class description. Two seperate collision detection methods are implemented to cater for interactions between the moving objects, being asteroid-to-planet, and finally bullet-to-asteroid. The algorithm checks each element in the npc vector, bullet vector, and the planet object to determine their position and radius. The implemented method acts as the algorithm described in the design phase, shown by fig 3.7:



```
// detect bullets hitting the npc's
if(d <= (v_asteroid[a]->radius + player1->v_bullet[b]->rad)){

    PlaySound("resources\\explosion.wav", NULL, SND_ASYNC);

    cout <<"collision detected : Asteroid shot" << endl;
    v_asteroid[a]->regCollision();
    v_asteroid[a]->takeDamage();

    if(v_asteroid[a]->alive == false){
        v_ex.push_back(new explosions(v_asteroid[a]->radius,v_asteroid[a]->sx,v_asteroid[a]->sy,v_asteroid[a]->sz,3,1));
        v_asteroid.erase(v_asteroid.begin()+a);
        a = a - 1;
    }
```

*Fig 3.6*

## Planet

The representation of the primary objective is established through the planet class. This class simply stores attributes of the primary objective, the "home planet", such as its health and graphical properties. It contains an implementation of the code adapted from the imageloader class.

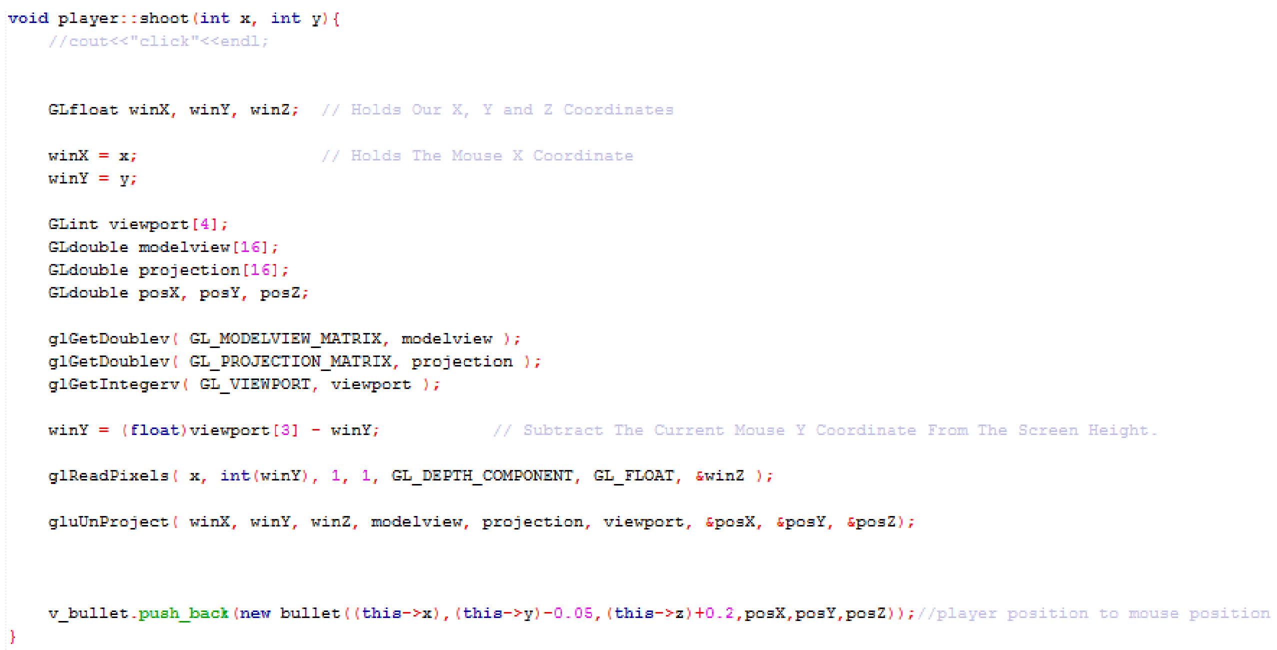
The planet class contains a render method that sets material properties and textures for a glutSolidSphere, and determines its size. An update function causes the planet to rotate slightly with time, to add to the visual effect (we really love that 8k texture).

Should a collision with the planet occur, the planet instance updates its "health" attribute to reflect that the collision has taken place. Beyond that, the planet class does not interact with any of the other classes

## Player

represented by a basic spaceship model in the game. The player class implements the model_obj class to import the .obj file that contains the spaceship model. The model was created using blender v2.73a, a graphics and modelling program available to the public.

The main responsibility of the player class involves the creation and control of bullet instances. Due to the fact that there is no limit to the number of bullets a player can create, it is necessary to store a vector of bullet pointers to keep track of each instance in memory. Each time the player clicks, the program creates a viewport and projects the bullet direction along a path relative to the player location and the mouses coordinates on the screen. Fig3.8 elaborates on this method



```
void player::shoot(int x, int y){
    //cout<<"click"<<endl;


    GLfloat winX, winY, winZ;  // Holds Our X, Y and Z Coordinates

    winX = x;                  // Holds The Mouse X Coordinate
    winY = y;

    GLint viewport[4];
    GLdouble modelview[16];
    GLdouble projection[16];
    GLdouble posX, posY, posZ;

    glGetDoublev( GL_MODELVIEW_MATRIX, modelview );
    glGetDoublev( GL_PROJECTION_MATRIX, projection );
    glGetIntegerv( GL_VIEWPORT, viewport );

    winY = (float)viewport[3] - winY;           // Subtract The Current Mouse Y Coordinate From The Screen Height.

    glReadPixels( x, int(winY), 1, 1, GL_DEPTH_COMPONENT, GL_FLOAT, &winZ );

    gluUnProject( winX, winY, winZ, modelview, projection, viewport, &posX, &posY, &posZ);



    v_bullet.push_back(new bullet((this->x),(this->y)-0.05,(this->z)+0.2,posX,posY,posZ));//player position to mouse position
}
```

*Fig 3.7*

## Point

this class was implemented in the early stages of the project. It is implemented in multiple classes to simplify storing the 3-dimensional coordinates of each instance of an element.

## World

As elements are not restricted to a 2-dimensional plane of movement, it becomes necessary to create a "boundary" to catch any elements that no longer reside within the allocated game space and remove them from memory. Whilst this class does not perform those actions, it does represent the boundary graphically. The world class is instantiated inside the physics_engine class, where the size values can be used to determine objects that leave the assigned area.

The inside of the sphere has been wrapped with a high definition texture to provide a backdrop that fits into the theme of the project.

## Discussion

The implementation of each of the classes to achieve the final product has a strong impact on any potential expansions that may be implemented in the future. Each of the classes does experience a variable degree of coupling or interconnectedness of a dependant nature, however the use of the physics_engine class as a central implementation space for all classes allows for many additions to be made.

In the event that a single class is removed, a majority of the other classes would continue with their full range of functionality. The only exclusions from that would be the interface classes, such as physics_engine, player, and explosion – each of which is/has some dependant class. However, adding additional objects to the project would require a fairly simple implementation in physics_engine. This leaves room for future expansion and class reusability, should the project be expanded upon.

One notable feature that impedes expansion is the collision detection. Although its implementation in this project covers all existing collision types, a seperate class will require its own method of collision detection to be designed and worked on. This coule be alleviated by using a "bounding box" class  to detect collisions between each box, which could be shaped to wrap each component and allow for further extension. Should a later extension require another object to be loaded in and textured, this can easily be achieved using the model_obj and imageloader classes to read in the information regarding the objects and  textures from a resource file. These could then be bound to a class object, and imlpemented in physics_engine.

Using GitHub proved to be a very useful tool whilst working on the project. It functioned as the team's workspace and allowed for easy code sharing. Each member could work in their own branch, developing a particular facet of the overall project goals. Once complete, the group could discuss pushing the altered segment to the "main" branch, which formed the final product, or discard a particularly buggy method.

Tasks were identified from the design document and assigned to each member to ensure that there was no duplication of effort. Wherever larger tasks appeared, multiple members could work on the section at the same time, and coordinate through either the WhatsApp group, or directly through the GitHub repository. Once complete, the members would either move on to their next assigned task, or assist with the next critical section of the code. This helped to achieve concentrated efforts on each of the tasks, minimising downtime

State

- Tasks were divided up and assigned to each member to ensure that no duplication of effort occurred. Each member was required to develop their assigned section in their branch, and once a working version was produced they either moved on to the next assigned section, or stated their completion to the group. This allowed for concerted efforts – very little time went to waste.

- The improvements that could be made for future projects definitely sits in the design phase – each phase of development occurred after large chunks of the project was already coded and working. No formal documentation was kept up to date between these phases, and editing only occurred at each milestone. This meant that progress was often repeated or left out, requiring checks and comparisons with earlier versions of the project – a very time consuming process.

Conclusion

- Overall, the desired objective was obtained. However, there were a few elements that did not make it to the final product.
    - The GLUT particle system was planned to create a visual effect for asteroid disintegration. However, other elements were deemed greater priority in order to achieve the greatest effect in the allotted time
    - ray casting was agreed upon to implement extended lighting effects, as well as enhancing events such as "firing"

evalutaion
future direction
achievements vs expectations
problems encountered

End matter
Bibliography
appendicies