

**Part I (code), Pick 2 Demonstrations:**

**The following questions are what I chose to do for both demonstrations, please see solutions file for my demonstrations.**

**Question 3:**

A demonstration of the nature of articulated motion. As a minimum, this will display a composite object built of two components, a base and arm. It is acceptable that these be simple coloured boxes. At a minimum, the base should translate, and both should both rotate, so that the arms is moved relative to the base. Additional marks will be awarded for the inclusion of additional components (so an upper and lower arm, with the upper arm connected to the lower arm with an “elbow”), and for the presence of interactive movement (preferably via the mouse).

**Question 4:**

A demonstration of collision detection and interactive object control via a small 3D dodging game. A “floor” should move towards the viewer, on which a player cube should be located. At random intervals, “obstacles” (also cubes) should appear from behind a back wall and move towards the viewer (as a consequence of the floor moving). The player is required to move forward and get to the back wall without hitting any obstacles (it is possible to dodge or jump them). Additional marks will be awarded for the inclusion of collision detection, randomization of the object shape, inclusion of a scoring mechanism (displayed via text) and inclusion of speed-ups.

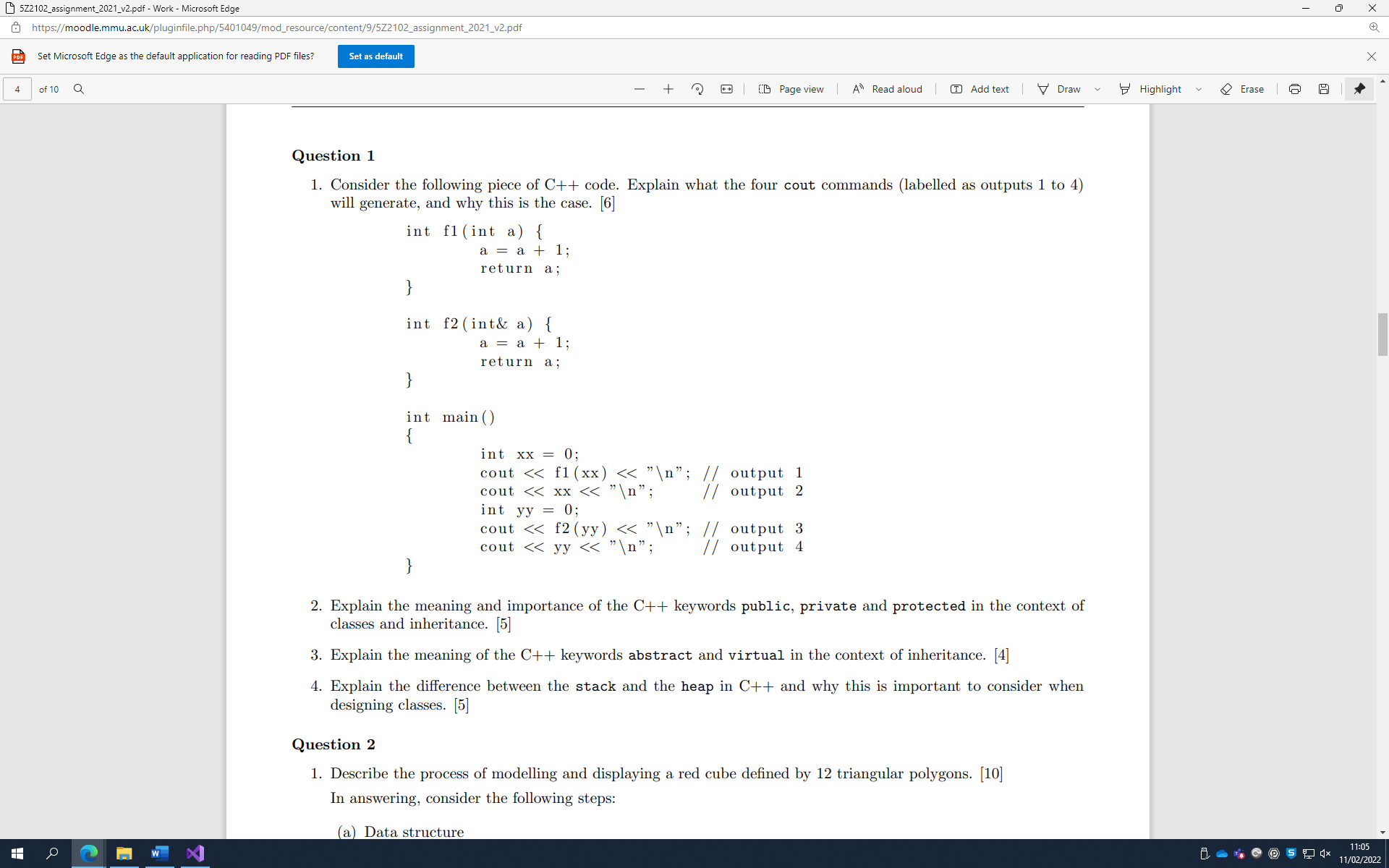
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Part II (Video, YouTube Link):**

<https://www.youtube.com/watch?v=GUWScSZHekw>

Sidenote: My video is terribly long but I forgot to add for the arm (Q3/OpenGL\_1) that I’m saving the scale matrix separately, so the scale operations happen independently of the compound transform.

So, in the draw function how it times all the cube matrices one after the other (line 150) is how the arm follows up from the base to the arm.

**Part III (text), Pick Three Questions:**

**Answer (Q1, 1.):**

The code in Question 1 will return as follows:

output 1 = 1,

output 2 = 0,

output 3 = 1,

output 4 = 1

The first output is 1 because we’re asking for the cout of ‘f1(int a)’, with ‘xx’ passed in. That function takes the amount ‘xx’ and adds 1 to it before returning the value. ‘xx’ is 0 to begin with, 0 + 1 = 1, so it will output 1.

The second thing in main being asked for is the cout of ‘xx’, that is 0, nothing is being changed by function f1, so it will just output 0.

The third output asks for f2(int& a), with ‘yy’ passed in. ‘yy’ was set also set to 0, just like ‘xx’, but the f2 function passes in a reference to the integer ‘a’, (aka “int***&*** a”), so the integer ‘yy’ will be modified along with ‘a’. The function returns the value of ‘a’. So output 3’s output will be 1, as 0 + 1 = 1.

The fourth output will be 1. Now that ‘yy’ was changed when the previous f2 function added 1 to it, so printing ‘yy’ will print the new value of 1, instead of 0.

**Q1, 2: Explain the meaning and importance of the C++ keywords public, private and protected in the context of classes and inheritance. [5]**

Public in classes: Public within classes means that the variable/function/etc can be accessed anywhere in the program.

Public in inheritance: Any inherited public members will be public and accessible in any derived classes.

Private in classes: Private within a class means that the variable/function/etc cannot be viewed or used outside of that specific class.

Private in inheritance: Private members will be made private in any derived classes, in turn making them inaccessible to them.

Protected in classes: Protected in classes are like private members but the variable/function/etc are only accessible to the class itself and any derivative classes (children).

Protected in inheritance: Any inherited protected members will be protected and public for itself and any derived classes.

**Q1, 3: Explain the meaning of the C++ keywords abstract and virtual in the context of inheritance. [4]**

C++ doesn’t have the keyword ‘Abstract’\* but to communicate “abstract” C++ uses something called ‘pure virtual functions’, as an Abstract class is any class with a pure virtual function. With the context of inheritance, abstract classes are useless without the implementation of it (cannot be instanced), as abstract classes aren’t implemented but are built upon or have features added to them via derived classes and inheritance.

For example: with the class ‘pet’ from the labs, all pets in this case needed to know how to make sounds but the pet class itself assigned sound to 0, and in any derived class such as ‘dog’ or ‘cat’, sound was then declared with their own appropriate value (such as “woof” and “miao”) that sound would point to or reference when ‘speak’ would be called.

The keyword ‘Virtual’ in C++ is different. When a virtual is declared in a class it will also be virtual in any derived classes. If you want to override a method it needs to be declared as virtual first, so in the pet class sound method, first assigned 0, when the dog or cat class is called the speak method will be overridden by the sound appropriate value (such as “woof” and “miao”).

\* While the ‘abstract’ keyword does not exist in C++, A Microsoft extension exists whereby the syntax used to declare a virtual function (eg ‘ = 0’) can be replaced with the keyword ‘abstract’.

**Q1, 4: Explain the difference between the stack and the heap in C++ and why this is important to consider when designing classes. [5]**

The Stack is a kind of algorithm using “first in, last out” (FILO), the first item to be added to the stack will be the last one to leave due to how they’re created. The stack is used implicitly by creating variables in functions and is freed implicitly by object scope (the ending of functions). Stacks are used by default by a class for its data members and is typically accessed by value.

The Heap is a kind of algorithm using random access. It is explicitly used with new/malloc (memory allocate: a function in C/C++ that allocates a block of memory from the heap) and is freed using delete/free. Accessing from a heap is typically done via pointers and heap is not the default classes use, this is important when considering designing classes as it means that the memory in a class must be deleted by the destructor and copy constructors must explicitly copy any associated data allocated to the heap.

**Question 3**

**Q3, 1: What are the two main projection methods used in OpenGL? [6]**

**- Describe their purpose, principles, and differences.**

**- Draw a pair of pictures, which will illustrate the projection of an object using the two projection methods.**

The two main projection methods used in OpenGL are ‘Orthographic (parallel)’ and ‘Perspective’. The Orthographic projection method causes all visible objects to be viewed at the same scale, unlike the Perspective projection method which gives a more realistic view of the area from a human perspective as it includes a field of view (FOV) and limited peripheral vision. Perspective view includes more about depth, and an item that is further away will look smaller on the screen due to this, just like a person’s view would in a real-world setting.

Orthographic projection shows the measurements of an object and not just how it might look, so it would be used more for buildings and architects. An architect would also use perspective projection, but it would be more of how a building’s interior looks from specific vantage points rather than how the building looks like as a whole (Shreiner, 2010).

**Diagram

Description automatically generated**

**Q3, 2: Describe the Phong reflection model, explaining why it is necessary to include an ambient lighting term in the model. [6]**

The Phong Reflection Model is a basis for lighting that has a relatively accurate rendering to what you would expect in the real world because in real life light comes from every direction on to a surface. The Phong Model uses four vectors (I, n, v, r) to determine the colour for each point (p) on a surface. Depending on if a surface is flat or curved, each vector will change depending on the point (p). The vector (l) is the light source, vector (n) is the normal of the point (p), vector (v) is the direction of the viewer, and vector (r) is the reflected ray, which is determined by vector (n) and vector (l). It is necessary to use ambient lighting as it diffuses at a constant rate and is a simple model for indirect lighting.

Using ambient light is an effective way of increasing the realism in any scene (Akenine-Moller, T. et al, 2008). Ambient reflections will ‘diffuse light’, rather than being a direct light. An example of this would be the lighting through another object such as a cloth or slightly translucent material. A scene with no indirect lighting would cause black/pitch dark sections whereas ambient light simulates light bouncing off other parts of the scene, for example on a cloudy day there is no direct sunlight due to clouds diffusing it, the cloud is absorbing the light and readmitting it equally in all directions. The world is not in pitch dark, but it is slightly darker than what one would expect on a sunny day and including this level of detail to a scene is adding ambience. Another example of an ambient lighting term could be a red light in a white room, as the bulb being switched on causes a red ambient term but when the light bulb is turned off, the room will return to looking white again. The room walls are not red and will never be red unless physically changed, but it looks red due to the ambient term.

**Q3, 3: Explain the four basic types of light sources in OpenGL [4].**

The four basic light sources in OpenGL are Ambient, Point sources, Distance/Parallel lighting, and Spotlighting. Ambient lighting redistributes lighting uniformly and depends on intensity, it is a level of light that effects everything by giving an approximation (accounts for anything that might hinder light, even slightly, such as dust particles, etc). Point light sources will emit light to all directions equally out from a single point. Distance light sources only have a vector and are a light source from infinitely far away, such as the light from the sun and are ever present in a scene. Distance lighting is also known as parallel lighting as the source is so far away that the light emitted is pretty much in parallel lines. Spotlight light sources are narrow areas of lights, as you would expect with a spotlight facing downwards, the source would be emitted from one starting point and outwards in a cone shape.

**Q3, 4: Describe the differences in purpose of the fragment-shader compared with the vertex-shader in OpenGL. [4]**

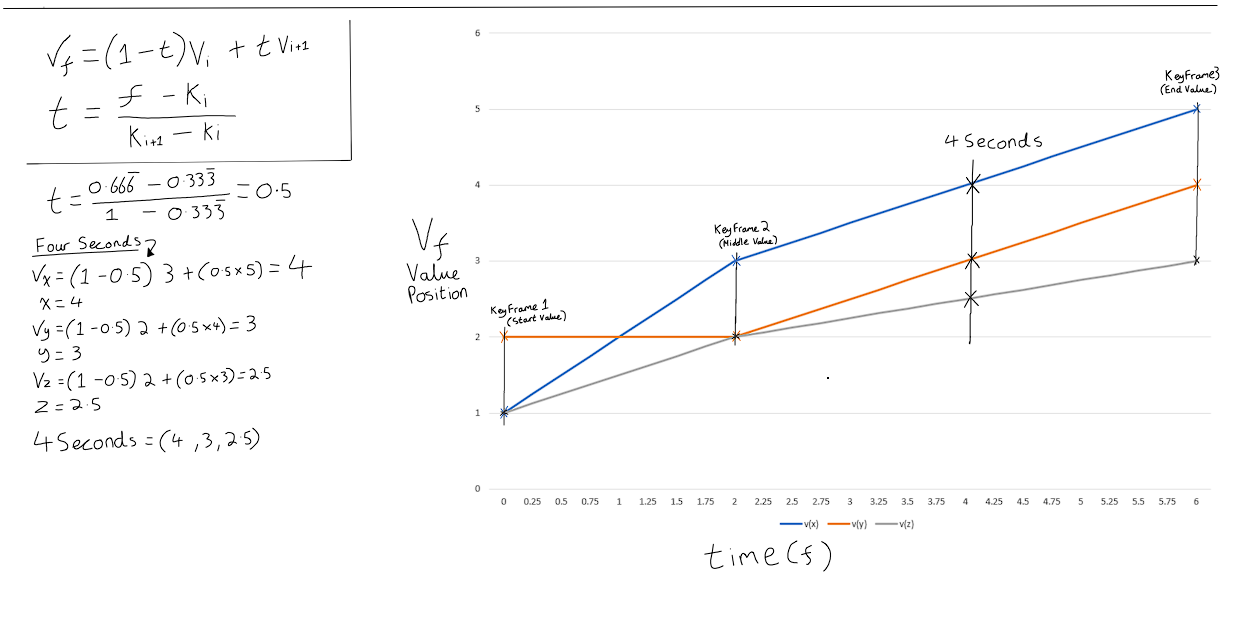
The main difference between Fragment shading and Vertex shading is that whiles fragment shaders work on pixels (“fragments”), vertex shaders work on vertices. Generally speaking, when you render an object, you can expect the vertex shader to run once per vertex, but the fragment shader will run once per pixel in the framebuffer. This means a fragment shader will happen later in the pipeline from a vertex shader and that the fragment shader will run many more times than the vertex shader. Also, the output from the vertex shader will be processed in an intermediate step before being processed by the fragment shaders, which gets stored in a framebuffer, to either be sent to the display or another fragment shader as in a multi-pass fragment shader (eg cell shading).

**Question 6  
Q6, 1: Consider the following interpolation problem. An object starts centred at position (1.0, 2.0, 1.0) and moves to position (3.0, 2.0, 2.0) after 2 seconds and then to position (5.0, 4.0, 3.0) after another 4 seconds. The movement is controlled by the following equations:**

**A picture containing clock, antenna

Description automatically generated**

**where (vi, ki) are the key value and fractional time for the i-th key-frame, and f is the fractional time. Draw a diagram showing its location against time, identify the type of interpolation used, and calculate its location 4 seconds after the start of the movement. [9]**



(Q6.1, ANSWER): The type of interpolation used is linear interpolation because it moves the same distance given any identical period of time. After 4 seconds the location will be halfway between key frame 2 and key frame 3 and as shown above the (x, y, z) positions will be at (4, 3, 2.5).

**Q6, 2: Write short explanations the following terms in the context of computer graphics [6]:**

**(6.2.a) Terrain heightmap**

Terrain height-mapping in the context of computer graphics is relatively simple. As one of

the two main techniques for terrain mapping, it uses separate files for Geometry and Texture. It is data driven and uses grids of triangles with an x and z plane, using the y values for the variety of height. The y values can typically be generated by using a colour channel or greyscale. So, if set from 0 to 1 or 256 by 256 vertices, making the y be set to 0 would cause the terrain’s plane to be completely flat. As the top of items would be lighter than the lower, shaded parts (top of a mountain compared to the lower sides of a mountain for example), the bitmap would reflect on this making any of the lighter parts be taller than a darker counterpart, imagining the light source being above the object, typically being closer to the light source will cause the item to be brighter.

Terrain in general will typically imply a natural, outdoor scenery such as coastlines, rivers, rockery, and vegetation. The use of a heightmap for terrain can improve the quality of the scene as different terrain has different sizes and values. Grass is not flat on the plane but it is not as tall as a mountain, grass will also not be inside of any bodies of water (rivers, oceans, etc) so having blending textures on a mesh can give a detailed and realistic map without having to manually add each sandy plane or grassy plane and rocky plane to the overall scene. Using “Splatting” or a terrain generation system like L3DT or Terragen can look at the height of a section and apply textures accordingly, for example, if close to sea-level, or below sea-level add sand texture, but if it is between height A and B add grassy texture and if above height B use rocky texture as it’ll imply mountainside, and so on.

**(6.2.b) Multi-texturing**

Multi-texture mapping in computer graphics makes for a more realistic ambience. Using two

or more textures to create details to a scene such as shadowing in a room corner with a diffuse texture and light map texture. As if there was no multi-texturing in a scene then a room with identical brightness inside the cave compared to outside of the cave would look unrealistic and feel very fake and break the viewer’s immersion, kind of an “uncanny valley” sort of situation as it looks like a cave, but something is off about the cave as it doesn’t align with real-world expectations. The texture of any object will have a difference in feel (roughness, smoothness, etc), show shiny it is, how transparent it is, and so on. Multi-texturing needs to account for each item and its own independent variations so that a cloth over a lamp with a lampshade will cause the scene in a room to look different (darkened, damper) to the lamp with a lampshade and no covering on the other side of the room (unhindered light, spilling out from the openings of the lampshade). Multi-texturing is a combination of the vertices and pixels at the rasterization stage, taking the geometric primitives (transform, light and clipping information) with the nongeometric primitive pixel operations (bitmap, colour, surface parameters, environmental properties or normal).

**(6.2.c) Fractal terrain**

Fractal terrain in the context of computer graphics can be described as an algorithm that

causes self-similarity, a way to specify environments. Central idea is constructing fragmented structures with similarity of detail. Some parts of an object will exhibit some similarity to larger parts of an object such as a mountainside. True fractals will have an infinite level of detail (depending on the screen resolution).

Fractal generation has three characteristics: Initiator, generator, and the recursion iterator. Starting with the seed-based element of the fractal (the shape being worked on), the generator’s rules being applied (whether linear, non-linear, complex, deterministic, etc) with some subject of randomness, and the recursive drawing with parameters that are controlled by the generator, where it’ll call the function again after calling it. As an example, using a mountainside on a 2D terrain, the fractal will start with a horizontal line, find the mid-point of the horizontal line, adjust the heights/displace it by a random y amount, and repeat again with the new set of smaller horizontal lines, eventually causing a shape that goes ‘up and up’ or ‘up and down’ or ‘down and down’, and so on, creating a realistic mountain shape overall for the terrain.

**Q6, 3: Briefly describe a use to which fractal geometry can be put in computer games. [3]**

Fractal Geometry can be put in computer games when something is needed that has detail from a distance as well as close-up. Fractal Geometry is typically used for things like natural objects, for example leaves, trees, snowflakes, coastlines, terrains, and anything that has a self-similarity. It is best used to create natural objects or phenomena that cannot be made by Euclidean Geometry, such as nature scenery, weather particles, and soft bodies of material like cloth.

Typical in games a fractal would be used for particle animation, if a character has a flame thrower, then the flames need to be realistic as not to break immersion. Using procedural modelling (PM) will cause the fire particles to spread out from a certain point over time and close enough but spread out at random short intervals, will look like a real-world flame being thrown out of a tip of a flame thrower gun barrel. Particle modelling (PM) can also be used for static scenery with moving parts, such as a waterfall, where the waterfall doesn’t move from the location of where it is based but water will look to be continuously flowing down from the highest point to the body of water below it. A con of fractals is its inability to model any precise object, but this works to an advantage with items such as water falls as it will give an example of a category (a water particle travelling downwards at random but downward facing angles), with ideally infinite detail, which does a very good job of being close to a real-life particle due to waterfalls themselves having an infinite number of water particles constantly always travelling down.

Stochastic Fractals can also be used in computer games to create realistic scenery for things such as a forests, trees, and leaves, showing a similar looking landscape with no identical components but still recognisable as a collection of those objects. Random values will give some leaves a different imperfect “oval” shape than others which correlates with the real world and where you can have a different random value at each step or a same generator with random parameters, it is good to use both for the best results as it can show a big change occurring every so often so one tree can be dramatically taller or have a twig much longer than its neighbouring trees, but it is still certainly a tree and similar enough to not be classified as a different object altogether.

**REFERENCES**

Angel, E. (2009) *Integrated Computer Graphics – A Top-Down approach Using OpenGL*. Fifth Edition., United States of America, Pearson Education, Inc.

Shreiner, D. (2010) *OpenGL Programming Guide – The Official Guide to Learning OpenGL, Versions 3.0 and 3.1.* Seventh Edition., United States of America, Pearson Education, Inc. *(pp: 132-133)*

Akenine-Moller, T., Haines, E., Hoffman, N. (2008) *Real-Time Rendering*. Third Edition., Wellesley, Massachusetts, A K Peters, Ltd.