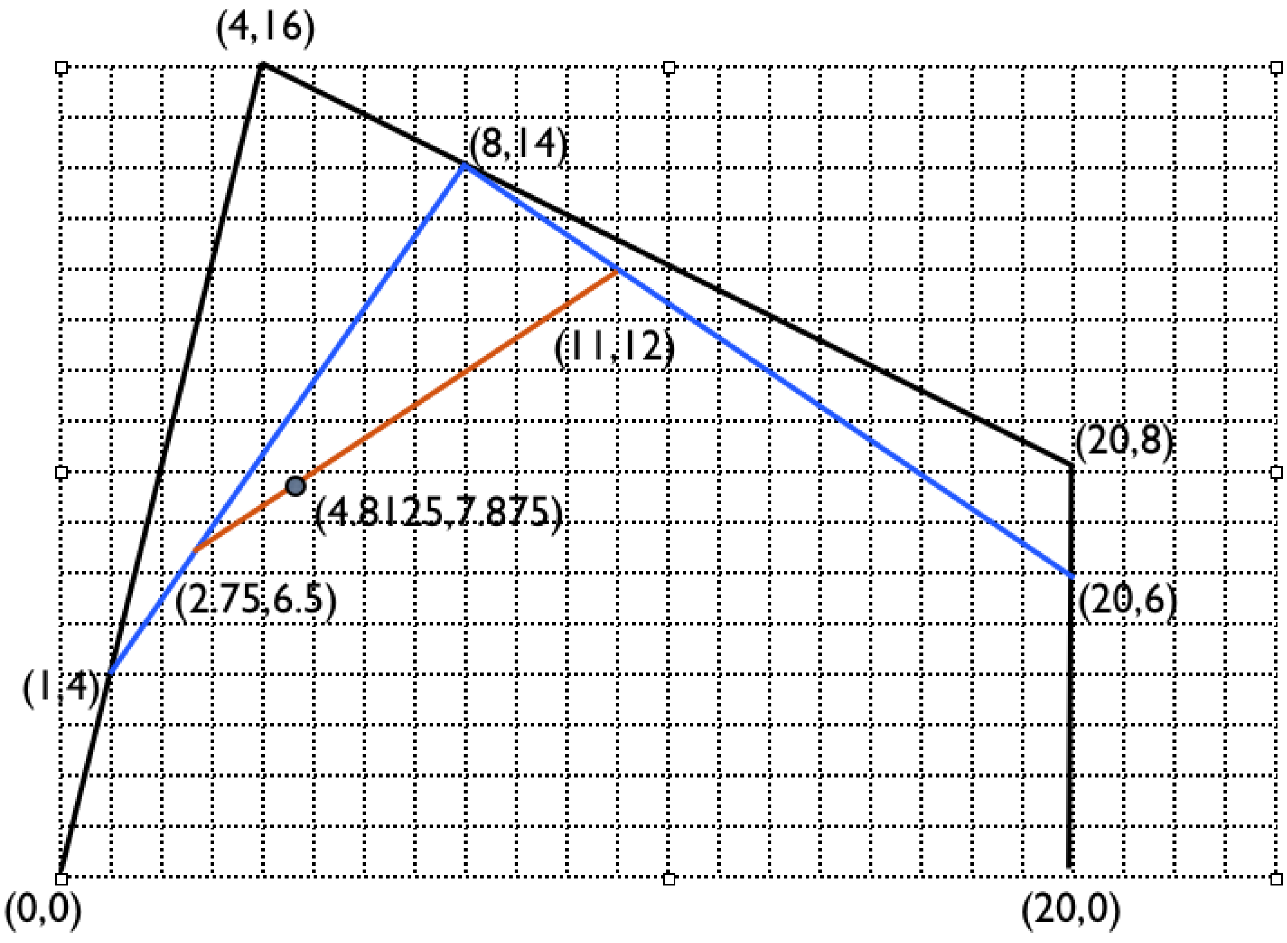
Week 7 Tutorial Solutions

Question 1: Bezier Curves

a. Use **de Casteljau's algorithm** to generate the point a **t = 0.25** on the degree 3 Bezier curve with control points (0,0), (4,16), (20,8), (20,0).

The point is (4.8125, 7.875) as illutrated by the construction below.



b. Prove that **de Casteljau's algorithm** describes the same curve as given by the degree 3 **Berstein polynomials**.

Prove that **de Casteljau's algorithm** describes the same curve as given by the degree 3 **Berstein polynomials**.

$$\begin{aligned}
P_{01}(t) &= (1-t)P_0 + tP_1 \\
P_{12}(t) &= (1-t)P_1 + tP_2 \\
P_{23}(t) &= (1-t)P_2 + tP_3 \\
P_{012}(t) &= (1-t)P_{01}(t) + tP_{12}(t) \\
&= (1-t)((1-t)P_0 + tP_1) + t((1-t)P_1 + tP_2) \\
&= (1-t)^2P_0 + 2t(1-t)P_1 + t^2P_2) \\
P_{123}(t) &= (1-t)^2P_1 + 2t(1-t)P_2 + t^2P_3) \\
P(t) &= (1-t)P_{012}(t) + tP_{123}(t) \\
&= (1-t)((1-t)^2P_0 + 2t(1-t)P_1 + t^2P_2)) + t((1-t)^2P_1 + 2t(1-t)P_2 + t^2P_3)) \\
&= (1-t)^3P_0 + 3t(1-t)^2P_1 + 3t^2(1-t)P_2 + t^3P_3 \\
&= B_3^0(t)P_0 + B_3^1(t)P_1 + B_3^2(t)P_2 + B_3^3(t)P_3
\end{aligned}$$

c. Find the tangent vector for the point on the Bezier curve from part a.

The equation for the tangent is

$$\begin{aligned}\frac{dP(t)}{dt} &= \sum_{k=0}^{m} \frac{dB_k^m(t)}{dt} P_k \\ &= m \sum_{k=0}^{m-1} B_k^{m-1}(t)(P_{k+1} - P_k)\end{aligned}$$

```
P1 - P0 = (4,16) - (0,0) = (4,16)
P2 - P1 = (20,8) - (4,16) = (16,-8)
P3 - P2 = (20,0) - (20,8) = (0,-8)

tangent = 3 * [(1-0.25)^2 (4,16) + 2*0.25*(1-0.25) (16,-8) + 0.25^2 (0,-8)]
        =  3 * [ (2.25,9) + (6,-3) + (0,-0.5)]
        = 3 * (8.25,5.5)
        = (24.75,16.5)
```

Question 2:

a. Find and fix the errors in the following shaders that are trying to do per fragment lighting calculations, considering only ambient light and diffuse reflections from one point light.

```
//Vertex Shader
in vec3 position;
in vec3 normal;

uniform mat4 model_matrix;

uniform mat4 view_matrix;

uniform mat4 proj_matrix;

in vec4 viewPosition;
in vec3 m;

void main() {
    vec4 globalPosition = model_matrix * vec4(position, 1);
    viewPosition = view_matrix * globalPosition;
    gl_Position = viewPosition;

    m = normalize(view_matrix*model_matrix * vec4(normal, 0)).xyz;
}

//Fragment Shader
out vec4 outputColor;

uniform vec4 input_color;

uniform mat4 view_matrix;

uniform vec3 lightPos;
uniform vec3 lightIntensity;
uniform vec3 ambientIntensity;

uniform vec3 ambientCoeff;
uniform vec3 diffuseCoeff;

in vec4 viewPosition;
in vec3 m;

void main()
{
    vec3 s = normalize(view_matrix*vec4(lightPos,1) - viewPosition).xyz;

    float ambient = ambientIntensity*ambientCoeff;
    float diffuse = max(lightIntensity*diffuseCoeff*dot(m,s), 0.0);

    float intensity = ambient + diffuse;

    outputColor = vec4(intensity,1)*input_color;
}


//Vertex Shader
in vec3 position;
in vec3 normal;
```

```
uniform mat4 model_matrix;

uniform mat4 view_matrix;

uniform mat4 proj_matrix;

out vec4 viewPosition; //These are outputs not inputs
out vec3 m;

void main() {
    vec4 globalPosition = model_matrix * vec4(position, 1);
    viewPosition = view_matrix * globalPosition;
    gl_Position = proj_matrix * viewPosition; //We need to multiply by the projection matrix here

    m = normalize(view_matrix*model_matrix * vec4(normal, 0)).xyz;
}

//Fragment Shader
out vec4 outputColor;

uniform vec4 input_color;

uniform mat4 view_matrix;

uniform vec3 lightPos;
uniform vec3 lightIntensity;
uniform vec3 ambientIntensity;

uniform vec3 ambientCoeff;
uniform vec3 diffuseCoeff;

in vec4 viewPosition;
in vec3 m;

void main()
{
    vec3 s = normalize(view_matrix*vec4(lightPos,1) - viewPosition).xyz;

    //The 3 values below should all be vec3 as they are RGB values
    vec3 ambient = ambientIntensity*ambientCoeff;
    vec3 diffuse = max(lightIntensity*diffuseCoeff*dot(m,s), 0.0);

    vec3 intensity = ambient + diffuse;

    outputColor = vec4(intensity,1)*input_color;
}
```

b. Half Lambert lighting is a technique designed to prevent the rear of an object (with respect to the light) losing its shape by only being lit by a constant ambient term.

In the standard lighting model, the dot product of the normal and light direction lies within the range -1 to 1 which then gets clamped to between 0 and 1 with the max function.

In Half Lambert shading we scale our dot product by 0.5 and then add 0.5 so it lies within the 0..1 range.

Modify the code above to implement this technique

```
//Fragment Shader
out vec4 outputColor;

uniform vec4 input_color;

uniform mat4 view_matrix;

uniform vec3 lightPos;
uniform vec3 lightIntensity;
uniform vec3 ambientIntensity;

uniform vec3 ambientCoeff;
uniform vec3 diffuseCoeff;

in vec4 viewPosition;
in vec3 m;
```

```
void main()
{
    vec3 s = normalize(view_matrix*vec4(lightPos,1) - viewPosition).xyz;

    //The 3 values below should all be vec3 as they are RGB values
    vec3 ambient = ambientIntensity*ambientCoeff;
    vec3 diffuse = max(lightIntensity*diffuseCoeff*(0.5 * dot(m,s) + 0.5), 0.0);

    vec3 intensity = ambient + diffuse;

    outputColor = vec4(intensity,1)*input_color;
}
```

c. Suppose we want to model light attenuation using the following equation

```
attenuation = 1/(1 + kd^2)
```

where k is an attenuation factor passed as a uniform into the shader and d is the distance netween the fragment and the light.

Modify the shaders to pass in a uniform variable k. Note: We only want to apply attenuation to the diffuse lighting calculations.

```
//Vertex Shader
in vec3 position;
in vec3 normal;

uniform mat4 model_matrix;

uniform mat4 view_matrix;

uniform mat4 proj_matrix;

out vec4 globalPosition
out vec4 viewPosition;
out vec3 m;

void main() {
    globalPosition = model_matrix * vec4(position, 1);
    viewPosition = view_matrix * globalPosition;
    gl_Position = proj_matrix * viewPosition

    m = normalize(view_matrix*model_matrix * vec4(normal, 0)).xyz;
}
//Fragment Shader
out vec4 outputColor;

uniform vec4 input_color;

uniform mat4 view_matrix;

uniform vec3 lightPos;
uniform vec3 lightIntensity;
uniform vec3 ambientIntensity;

uniform vec3 ambientCoeff;
uniform vec3 diffuseCoeff;

uniform float k;

in vec4 viewPosition;

in vec4 globalPosition;

in vec3 m;

void main()
{
    vec3 s = normalize(view_matrix*vec4(lightPos,1) - viewPosition).xyz;

    //The 3 values below should all be vec3 as they are RGB values
    vec3 ambient = ambientIntensity*ambientCoeff;
    vec3 diffuse = max(lightIntensity*diffuseCoeff*dot(m,s), 0.0);

    float d = length((vec4(lightPos,1) - globalPosition).xyz)
    float attenuation = 1.0 / (1.0 + k * pow(d, 2));
```

```
        vec3 intensity = ambient + attenuation*diffuse;

        outputColor = vec4(intensity,1)*input_color;
    }
```

Question 3:

Consider an L-System with the rules

```
X-> F[+X]F[-X]+X
F -> FF
```

Starting with the symbol `X` compute the first three iterations of this system.

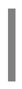
Suppose the symbols translate to the following commands:

```
X : null
F : forwards 10
+ : rotate 30
- : rotate -30
[ : push the current pen position
] : pop the pen position
```

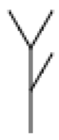
Draw the shapes for the strings generated above.

0	X
1	F[+X]F[-X]+X
2	FF[+F[+X]F[-X]+X]FF[-F[+X]F[-X]+X]+F[+X]F[-X]+X
3	FFFF[+FF[+F[+X]F[-X]+X]FF[-F[+X]F[-X]+X]+F[+X]F[-X]+X]FFFF[-FF[+F[+X]F[-X]+X]FF[-F[+X]F[-X]+X]+F[+X]F[-X]+X]+FF[+F[+X]F[-X]+X]FF[-F[+X]F[-X]+X]+F[+X]F[-X]+X

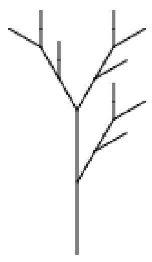
Using the Madflame L-system app, the 1st generation looks like this:



Using the Madflame L-system app, the 2nd generation looks like this:



Using the Madflame L-system app, the 3rd generation looks like this:



Assignment 2

If there is any time left discuss assignment 2.