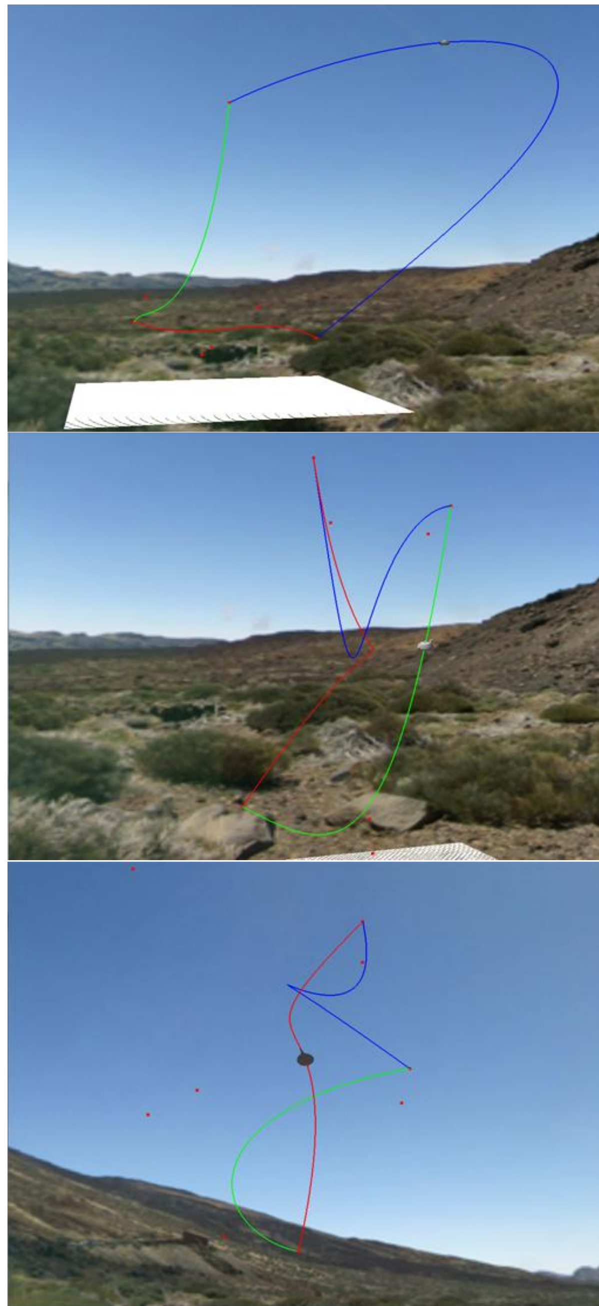
CE1051A Bézier Curve Coursework Report

By Vladeta Stojanovic (0602920@live.abertay.ac.uk)



Introduction

This report provides an overview concerning the application of key methods of mathematical modelling techniques for Bézier curves in real time 3D. The coursework specification required the implementation of rendering and modelling 3D object motion along parametrically defined Bézier curves. The coursework application was developed using the OpenGL 1.2 graphics API, along with the GLUT 3.7 OpenGL wrapper library for handling Win32 windowing functionality and user input.

Coursework Application Overview

The developed coursework application allows the generation and modelling of a parametric rollercoaster model, defined by a set of three different third-order cubic Bézier curves (with 12 control points in total). The Bézier curve generation applied in the coursework application makes use of De Casteljau's algorithm for numerically evaluating the Bézier curves. The Bézier curves themselves are modelled on the basis of Bernstein polynomials, where each of the control points of the parametric curve has influence on all other control points defining the given curve. Unlike Bézier splines, where the gradients of each of the control points can be evaluated independently, the applied curve generation method used in the coursework application instead focuses on the traditional representation of Bézier curves. The given rollercoaster shape is made up of three separate curves, which are joined together at their defined end-points. This allows for the generation of rollercoaster style parametric curve shapes.

The coursework application allows the user to examine the generated Bézier curves in 3D space, and also allows the user to examine the motion of a 3D object as it moves along the curve. A default curve is generated at the start-up of the program, based on control point values read in from three separate text files. Additionally, the program allows the user to infinitely generate new curves based on randomly generated control point values. Linear interpolation based transitional generation is applied, allowing the curves to smoothly "morph" from their current shape into their new shape, based on the altered control point values. The principals of potential to kinetic energy transfer are also modelled in real time 3D. This allows the user to see how the given 3D object in the scene moves along the curve with varying velocities, based on the calculated slope value of a given gradient tangent on the curve. With the addition of gravitational force, this allows for the modelling of rollercoaster motion of the 3D object as it moves along the curve. However, this system only works for the default curve, as for all other randomly generated curves the motion of the 3D object along the curve makes use of constant velocity. This is because the velocity values at given slopes for the default curve are defined manually.

Additionally, the coursework application allows the user to examine the 3D scene using three different viewing modes: Free-view 1st person mode, 3rd person object-centric mode and 1st person rollercoaster viewing mode.

Overview of Implemented Bézier Curve Modelling Methods

The parametric vector equation of a line in 2D or 3D space is defined as [Vince 2006][Dempski 2003]:

$$\underline{\boldsymbol{r}}(t) = (1-t)\underline{\boldsymbol{a}} + \underline{\boldsymbol{b}}(t)$$

The given parametric line is defined as having a minimum of two control points (so technically it's a straight line), where the total length of the line, denoted as t, is in the range from 0 to 1. This allows for obtaining the coordinate values along the line based on the parametric Cartesian curve definition. The number of control points for a given curve is usually approximated on a quadratic or cubic basis, meaning that the given curve usually has one or two additional control points used to define its shape in 2D or 3D space. This is known as the *order* of the curve. Higher order curves (those above fourth order), are usually avoided as the numerical approximation of a given 2D or 3D curve becomes more computationally expensive as the number of control points increases.

From the parametric line definition above making use of a starting and ending point definition, a second-order quadratic curve definition is obtained using simple binomial expansion, where the new parametric curve definition can be defined as [Vince 2006]:

$$\underline{\boldsymbol{r}}(t) = (1-t)^{2}\underline{\boldsymbol{a}} + 2t(1-t)\underline{\boldsymbol{b}} + t^{2}\underline{\boldsymbol{c}}$$

Where for a given set of control points, the 2D or 3D coordinate values along the curve can be obtained at point t. This allows for the computation of a numerically approximated shape of a given parametric curve in 2D or 3D space. The implemented curve modelling methods featured in the coursework application are used to approximate only third-order cubic Bézier curves, defined as [Vince 2006]:

$$\underline{\boldsymbol{r}}(t) = (1-t)^{3}\underline{\boldsymbol{a}} + 3t(1-t)^{2}\underline{\boldsymbol{b}} + 3t^{2}(1-t)\underline{\boldsymbol{c}} + t^{3}\underline{\boldsymbol{d}}$$

But just for clarity, it is important to note that a Bézier curve can have a general parametric form (with varying numbers of control points, thus varying degrees of complexity). This general form of the Bézier curve is expressed in the form of Bernstein polynomials, and is defined as [Vince 2006]:

$$\underline{\boldsymbol{B}}_{n}(t) = \sum_{\mathrm{r}=0}^{\mathrm{n}} \underline{\mathbf{b}}_{\mathrm{r}}\, B_{r,n}(t)$$

$$\text{Where } B_{r,n}(t) \equiv \frac{n!}{r!(n-r)!}\, t^{r}(1-t)^{n-r}$$

It is evident from the definition above of the Bernstein polynomial form of a Bézier curve that approximating higher order curves requires a lot of recursive computation. Thus for most applications concerning 2D or 3D interactive graphics rendering, second or third order Bézier curves are sufficient. However, even lower degree Bernstein polynomials are

inefficient to compute recursively without optimization, so a numerical algorithm exists that can evaluate Bernstein polynomials using a defined stepping size for interpolation accuracy.

This algorithm is called De Casteljau's algorithm. De Casteljau's algorithm works by recursively subdividing the curve tangents formed by connecting the control points of a given Bézier curve. The amount of subdivision performed is relative to the defined stepping size, which in this case is the value of *t*. This means that the subdivision of the curve (thus its approximation accuracy), depends on the precision of t that's sought after. De Casteljau's algorithm is based on the recurrence relation to a given cubic Bézier curve, as defined as [Farin 1995][Vince 2006] [Slater et al 2002]:

$$\underline{\boldsymbol{B}}(t) = \sum_{i=0}^{n} \underline{\boldsymbol{B}}_i \underline{b}_{i,n}(t)$$

Where the Bernstein polynomial *b* can be evaluated as:

$$\underline{\boldsymbol{b}}_i^{(j)} = (1-t)\underline{\boldsymbol{b}}_i^{(j-1)} + t\underline{\boldsymbol{b}}_{i+1}^{(j-1)} \quad (j = 1, 2, 3 \ldots n \text{ and } i = 0, 1, 2, 3 \ldots n - j)$$

Where

$$\underline{\boldsymbol{B}}(t_0) = \underline{\boldsymbol{B}}_0^{(n)}$$

A simple way to explain De Casteljau's algorithm, in terms of evaluating a cubic Bézier curve, is to first take note of the four control points that make up the cubic Bézier curve, defined as:

$$P_0\,; P_1; P_2; P_3$$

Between these four points, three new mid points are generated using linear interpolation, and these points are denoted as:

$$Q_0; Q_1; Q_2$$

Where

$$Q_0 = (1-t)P_0 + P_1(t);\ Q_1 = (1-t)P_1 + P_2(t);\ Q_2 = (1-t)P_2 + P_3(t)$$

The above equations linearly interpolate the mid-point values of the four main control points of the cubic Bézier curve, up to a 2nd polynomial degree. The three interpolated midpoints are further evaluated to a 3rd polynomial degree, producing two new mid points, denoted as:

$$R_0;\ \ R_1$$

Where

$$R_0 = (1-t)\big((1-t)P_0 + P_1(t)\big) + t((1-t)P_1 + P_2(t)\ )$$

$$R_1 = (1-t)\big((1-t)P_1 + P_2(t)\big) + t((1-t)P_2 + P_3(t)\ )$$

So with the above recursive relations described, the application of De Casteljau's algorithm for evaluation of a cubic Bézier curve can be applied to a given sampling point situated on the generated curve, denoted as a vector $\underline{\boldsymbol{S}}$ (in this case $\underline{\boldsymbol{S}}$ is a 3D vector), where:

$$\underline{\boldsymbol{S}}_{\mathrm{x}} = (-t^3 + 3t - 3t + 1)P_0 + (3t^3 - 6t^2 + 3^t)P_1 + (-3t^3 + 3t^2)P_2 + p_3(t^3)$$

$$\underline{\boldsymbol{S}}_{\mathrm{y}} = (-t^3 + 3t - 3t + 1)P_0 + (3t^3 - 6t^2 + 3^t)P_1 + (-3t^3 + 3t^2)P_2 + p_3(t^3)$$

$$\underline{\boldsymbol{S}}_{\mathrm{z}} = (-t^3 + 3t - 3t + 1)P_0 + (3t^3 - 6t^2 + 3^t)P_1 + (-3t^3 + 3t^2)P_2 + p_3(t^3)$$

Where the above linear interpolation equation definitions for each of the coordinates of the point $\underline{S}$ are in expanded 3rd degree polynomial form. The above relations can also be described in the standard Bernstein polynomial form:

$$\underline{\boldsymbol{S}}_{\mathrm{x}} = \underline{\boldsymbol{B}}_0^3(t)P_0 + \underline{\boldsymbol{B}}_1^3(t)P_1 + \underline{\boldsymbol{B}}_2^3(t)P_2 + + \underline{\boldsymbol{B}}_3^3(t)P_3$$

$$\underline{\boldsymbol{S}}_{\mathrm{y}} = \underline{\boldsymbol{B}}_0^3(t)P_0 + \underline{\boldsymbol{B}}_1^3(t)P_1 + \underline{\boldsymbol{B}}_2^3(t)P_2 + + \underline{\boldsymbol{B}}_3^3(t)P_3$$

$$\underline{\boldsymbol{S}}_{\mathrm{z}} = \underline{\boldsymbol{B}}_0^3(t)P_0 + \underline{\boldsymbol{B}}_1^3(t)P_1 + \underline{\boldsymbol{B}}_2^3(t)P_2 + + \underline{\boldsymbol{B}}_3^3(t)P_3$$

Where,

$$\underline{\boldsymbol{B}}_0^3(t) = -t^3 + 3t^2 - 3t + 1$$

$$\underline{\boldsymbol{B}}_1^3(t) = 3t^3 - 6t^2 + 3t$$

$$\underline{\boldsymbol{B}}_2^3(t) = -3t^3 + 3t^2$$

$$\underline{\boldsymbol{B}}_3^3(t) = t^3$$

This thus provides relative proof of the statement $\underline{\boldsymbol{B}}(t_0) = \underline{\boldsymbol{B}}_0^{(n)}$. The value of *t* is taken as the stepping size along the curve. The value of *t* controls the generation accuracy (smoothness) of the approximated cubic Bézier curve. While this may seems a bit abstract, the implementation of De Casteljau's algorithm in terms of actual code is fairly straight forward. The numerically evaluated recurrence relation for each value of *t* is linearly interpolated between each of the four control points of the parametric curve.

The lerping function is implemented as [Pipenbrinck 1999]:

```
inline void Lerp(GLfloatPoint& dest, const GLfloatPoint& a, const GLfloatPoint&
b, const float t)
{
        dest.x = a.x + (b.x-a.x)*t;
        dest.y = a.y + (b.y-a.y)*t;
        dest.z = a.z + (b.z-a.z)*t;

}
```

Where each of the four control points of the Bézier curve are defined by the GLfloatPoint struct:

```
struct GLfloatPoint
{
       GLfloat x,y,z;
};
```

The Lerp() function is used to subdivide the current control point tangents by linearly interpolating the value of the new midpoints between each of the two main control points, and then subdividing the current mid points and placing new midpoints between them, and then subdividing those and so on. The Lerp() function is used to calculate the midpoints for a given cubic Bézier curve in the ComputeBézier() function implementation [Pipenbrinck 1999]:

```
void BCurve::ComputeBézier(GLfloatPoint& dest,const GLfloatPoint& a, const
GLfloatPoint& b, const GLfloatPoint& c, const GLfloatPoint& d, const float t)
{
    GLfloatPoint ab,bc,cd,abbc,bccd;
    Lerp(ab, a,b,t);
    Lerp(bc, b,c,t);
    Lerp(cd, c,d,t);
    Lerp(abbc, ab,bc,t);
    Lerp(bccd, bc,cd,t);
    Lerp(dest, abbc,bccd,t);
}
```

The input variables *a*, *b*, *c*, *d* each represent the four control points of the curve. The variables *ab*, *bc* and *cd* are the three midpoints between *a*, *b*, *c* and *d*. The variables *abbc*, *bccd* are the midpoints of *ab*, *bc* and *cd*. Therefore the ComputeBézier() function can approximate the shape of the curve by drawing the tangents between each of the mid points. Changing the value of *t* provides different smoothness results:

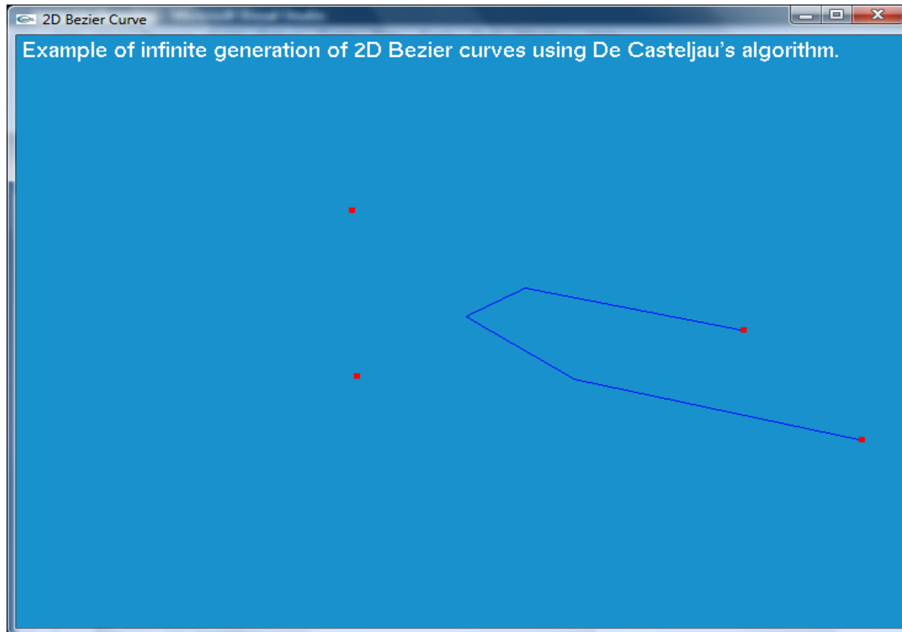


Fig 1: Approximation of a cubic Bézier curve using De Casteljau's algorithm, where the value of t is 1.

Increasing the value of *t* to 0.25 produces a smoother curve approximation:

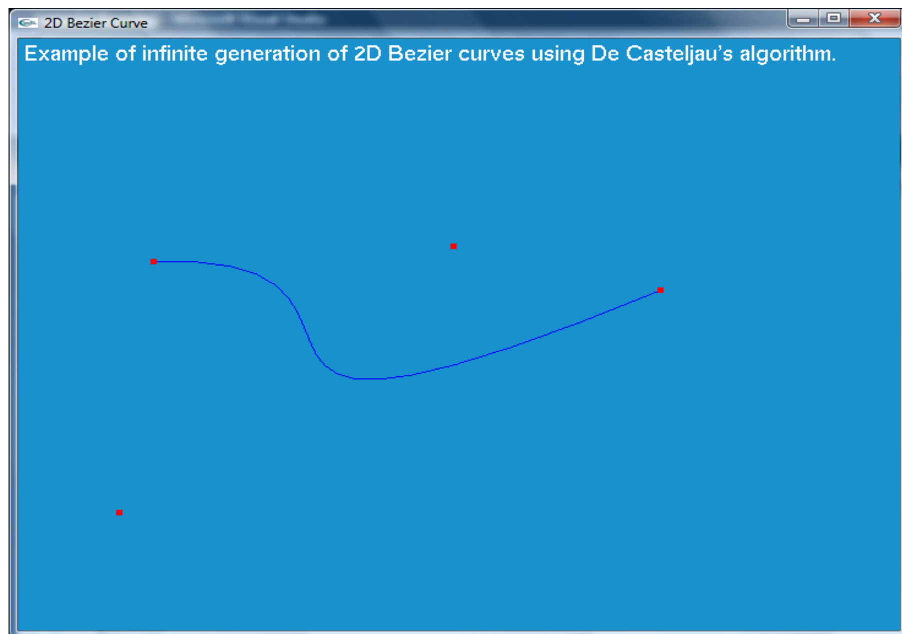


Fig 2: Approximation of a cubic Bézier curve using De Casteljau's algorithm, where the value of t is 0.25.

However, the value of *t* used for the approximation of Bézier curves in the actual application program is 1×10^{-3}, and is implemented in the following piece of code [Pipenbrinck 1999] [Miller 1998]:

```
void BCurve::DrawCurve(float r, float g, float b)
{
	glColor3f(r, g, b);
	glBegin(GL_LINE_STRIP);
	for (int i=0; i< 1000; ++i)
	{
		float t  = static_cast<float>(i)/999.0f;

		ComputeBézier(points[0], points[1], points[2], points[3], points[4], t);
		glVertex3f (points[0].x, points[0].y, points[0].z);
	}
	glEnd();
}
```

This allows for incredibly smooth approximation of cubic Bézier curves.

The final piece of functionality needed for generation of cubic Bézier curves is to be able to sample the value coordinates along the generated curve. This is important for implementing any sort of motion along the curve. The function `GLfloatPoint PointOnSpline(BCurve *bSpline,float t)` takes in a pointer to a given spline, as well as the stepping value *t,* and returns the new XYZ coordinate value to the sampling point assigned to the function. The PointOnSpline() function is a direct implementation of the cubic linear interpolation of the curve described above [Humphrey 2005]:

```
GLfloatPoint PointOnSpline(BCurve *bSpline,  float t)
{
	float t0, t1, t2;
	GLfloatPoint tempPoint;

	t0 = 1 - t;
	t1 = powf(t0, 3);
	t2 = powf(t, 3);

	tempPoint.x = t1*bSpline->points[1].x + 3*t*t0*t0*bSpline->points[2].x +
	3*t*t*t0*bSpline->points[3].x + t2*bSpline->points[4].x;

	tempPoint.y = t1*bSpline->points[1].y + 3*t*t0*t0*bSpline->points[2].y +
	3*t*t*t0*bSpline->points[3].y + t2*bSpline->points[4].y;

	tempPoint.z = t1*bSpline->points[1].z + 3*t*t0*t0*bSpline->points[2].z +
	3*t*t*t0*bSpline->points[3].z + t2*bSpline->points[4].z;

	return tempPoint;
}
```

Overview of Implemented Bézier Curve Motion Modelling Methods

The next important topic to discuss is the implementation of motion algorithms for movement of 3D objects along the generated cubic Bézier curve. The main generated curve shape featured in the coursework application is that of a rollercoaster. This shape is generated using three different Bézier splines and connecting the first and last points of each of the splines to the corresponding splines first or last point. This implemented in the following function:

```
void SetupCurves()
{
	gBCurve0->points[4] = gBCurve1->points[1];
	gBCurve1->points[4] = gBCurve2->points[1];
	gBCurve2->points[4] = gBCurve0->points[1];
}
```

The coursework application can also generate an infinite number of other closed spline shapes from the main rollercoaster shape (while showing the changes from the old coordinates to the new generated spline coordinates to the user in a smooth transitional manner, using linear interpolation). A 3D teapot object is placed in the scene and can move along any of these generated curves using constant velocity only. However, the 3D teapot can move using varying velocities along the default curve, by taking into account manually computed tangent gradients as it moves along the curve and simulating the transference of potential to kinetic energy (and vice versa), while also taking into account gravitational force. The definitions for potential and kinetic energy are [Neumann 2004]:

$$PE = mgY$$

$$KE = \frac{1}{2} mv^2$$

Where *m* is the mass of the teapot object (50kg), *g* is the gravity constant (9.8), *Y* is the current coordinate value of the local Y axis of the teapot model (not used in the implementation), and *v* is the current velocity of the teapot as it moves along the curve. In order to determine at what point on the rollercoaster to apply the transfer of energy of the object, the gradient value of the tangent slope sampled along the curve must be computed. This gradient slope value is denoted as *K*, and is defined as [Neumann 2004]:

$$K = \frac{Y[step + 1] - Y[step]}{X[step + 1] - X[step]}$$

Where Y and X are the current sampled point values along the curve, and *step* is the current stepping speed along the curve (how fast the point coordinate values are sampled along the curve). Thus rollercoaster motion can be modelled for the default curve, based on the approximated values of each of the gradient slopes along the curve.

The gradient slope values are recorded (by hand), and the application of PE or KE transference is applied at the estimated points along the rollercoaster curve shape. This is implemented in the CheckSlope() function (and is called every frame):

```
void CheckSlope(float dt)
{
	k = ((mtPointSet0.y + dt)  -  (step)) / ((mtPointSet0.x + dt)  -  (step));

	if(k > 2.5f)
	{
		step = Lerp((dt * (0.5 * gTeapot->mMass * 9.8f)) * 0.01f, step,
0.000001f); //PE
	}
	else if(k > 2.3f && k < 2.5f)
	{
		step = Lerp((dt * (0.5 * gTeapot->mMass * 9.8f)) * 0.01f, step,
0.00000005f); //PE
	}
	else if(k > 2.0f && k < 2.3f)
	{
		step = Lerp((dt * (0.5 * gTeapot->mMass * 9.8f)) * 0.01f, step,
0.00000003f); //PE
	}
	else if(k > 1.8f && k < 2.0f)
	{
		step = Lerp((dt * (0.5 * gTeapot->mMass * 9.8f)) * 0.01f, step,
0.00000002f); //PE
	}
	else if(k > 1.7f && k < 1.8f)
	{
		step = Lerp((dt * (0.5 * gTeapot->mMass * 9.8f)) * 0.01f, step,
0.000000015f); //PE
		rot = Lerp(0.5, -30, 30.0f);
	}
	else if(k > 1.6f && k < 1.7f)
	{
		step = Lerp((dt * (0.5 * gTeapot->mMass * 9.8f)) * 0.01f, step,
0.00000001f); //PE
	}

	else if(k > 1.4f && k < 1.6f)
	{
		step = Lerp((dt * (0.5 * gTeapot->mMass * 9.8f)) * 0.01f, step,
0.000000009f); //PE
	}
	else if(k > 1.2f && k < 1.4f)
	{
		step = Lerp((dt * (0.5 * gTeapot->mMass * 9.8f)) * 0.01f, step,
0.000000001f); //PE
	}
	else if(k > 1.0f && k < 1.1f)
	{
		step = Lerp((dt * (gTeapot->mMass * 9.8f)) * 0.01f, step, 0.0005f); //KE
		rot = Lerp(0.5, rot, -30.0f);
	}
	else if(k > 0.8f && k < 1.0f)
	{
		step = Lerp((dt * (gTeapot->mMass * 9.8f)) * 0.01f, step, 0.0006f); //KE
	}
	else if(k > 0.6f && k < 0.8f)
	{
```

```
                step = Lerp((dt * (gTeapot->mMass * 9.8f)) * 0.01f, step, 0.0009f); //KE
        }
        else if(k > 0.4f && k < 0.6f)
        {
                step = Lerp((dt * (gTeapot->mMass * 9.8f)) * 0.01f, step, 0.0007f); //KE
        }
        else if(k > 0.2f && k < 0.4f)
        {
                step = Lerp((dt * (gTeapot->mMass * 9.8f)) * 0.01f, step, 0.002f); //KE
        }
        else if(k < 0.2f)
        {
                step = Lerp((dt * (gTeapot->mMass * 9.8f)) * 0.01f, step, 0.0007f); //KE
        }
}
```

Finally, the local model axis rotation of the teapot as it moves along the curve is calculated as [Neumann 2004]:

$$\cos(\theta) = \frac{-kmg}{mg\sqrt{(1+k^2)}} = \frac{-k}{\sqrt{(1+k^2)}}$$

And this is implemented in the CalculateSlopeRotation() function, also called once per frame:

```
void CalculateSlopeRotation()
{
        float A;
        float B;

        A = -k;
        B = sqrtf(1 + powf(k, 2));

        theta = powf(cosf(A / B), -1);

}
```

Overview of Implemented Camera Modes

The final piece of functionality that was added to the Bézier curve coursework application was the implementation of three different camera models. Below are brief descriptions of each of the camera models:

1) *Free-view 1st Person Scene Camera:* This camera mode is the standard camera mode for most 3D scenes. It features six degrees of freedom, vector based movement along the local camera forward axis (as well as strafing along the local camera right axis), and XY local view rotations controlled by the users mouse.

2) *3rd Person Object Tracking Camera:* This is a more advanced camera mode, as it allows the camera to smoothly follow the 3D teapot object as it moves along the generated Bézier curve model. This is enabled with the use of linear interpolation to update the camera coordinates with the teapot as the viewing target, for every frame. The camera is kept at a fixed distance from the 3D teapot object, allowing for smooth 3rd person observation of the teapot model while it is in motion.

3) *1st Person View "Rollercoaster" Camera:* This camera model allows the user to examine the implemented motion control along the generated Bézier curve model, from a 1st person view. This camera mode is interesting as it gives the user a better feeling of how it is like to "ride" the generated rollercoaster Bézier curve model. It works best for the default rollercoaster model, as the velocity of the camera is updated with accordance to the implemented potential and kinetic energy transference corresponding to the calculated slope gradients along the generated curve.

For a more in depth look into the implementation techniques used for moving a camera along a curve, please see [Twigg 2003].

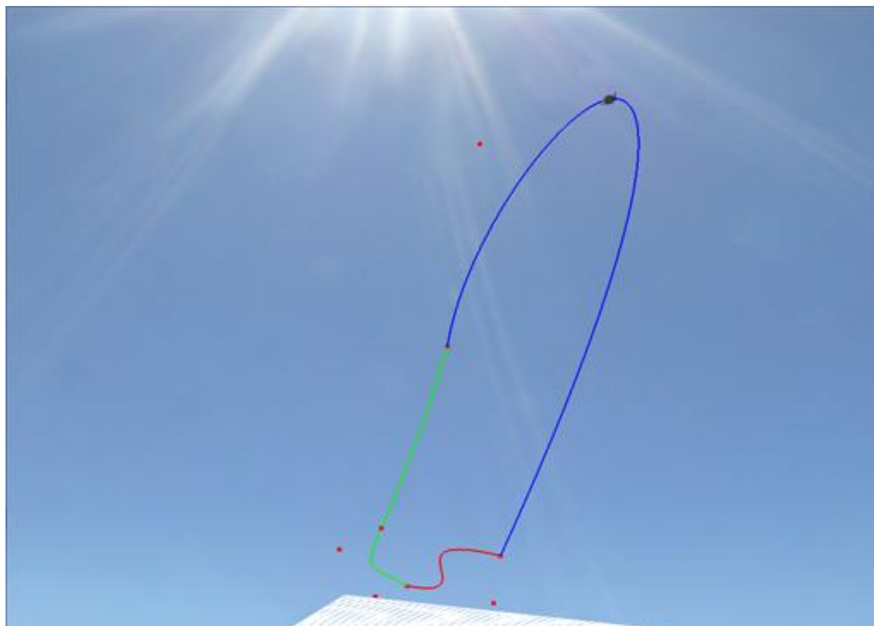


Fig 3: Example of the free-view 1st person scene camera mode.

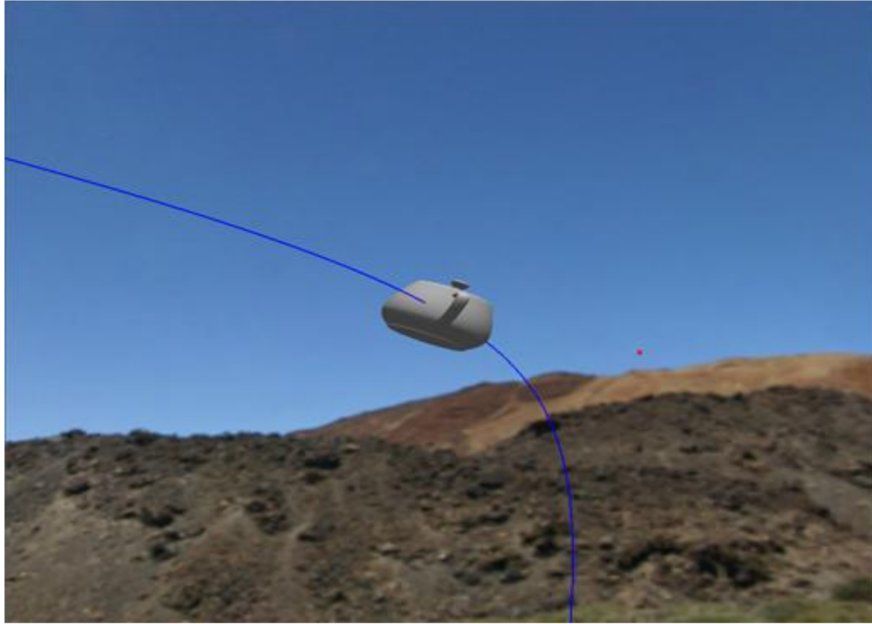


Fig 4: Example of the 3rd person object tracking camera mode.

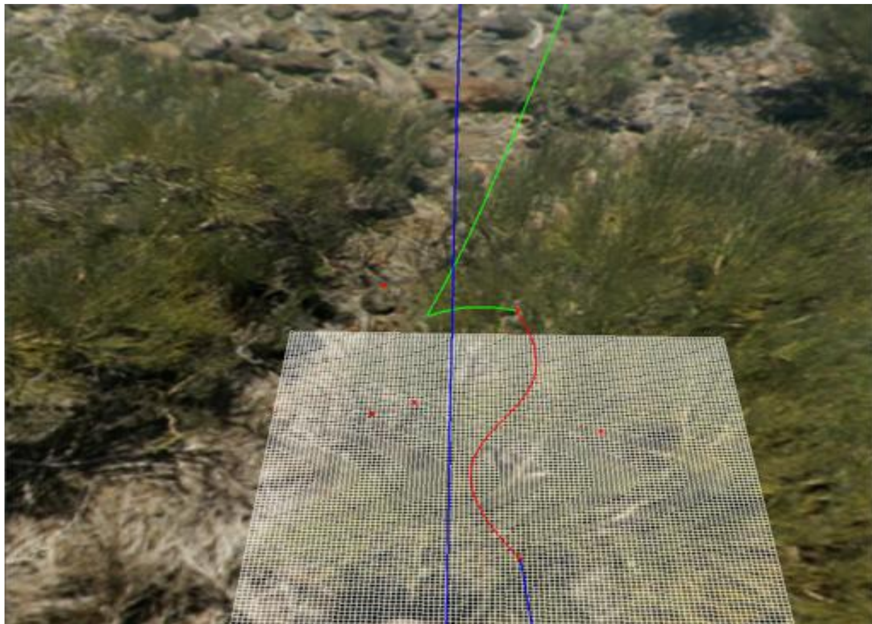


Fig 5: Example of the 1st Person view "Roller Coaster" camera mode.

Performance Considerations

The generation of 3D Bézier curves using De Casteljau's algorithm works fairly fast (the generation time is a few milliseconds). One of the main optimizations made to the implemented coursework application was the pre-calculation of the curve gradient slope values. In addition to the curve generation performance, the cubic interpolation point sampling for coordinate values along the curve is quite fast as well (considering the fact that the majority of the mathematics calculations make use of the standard Math.h mathematics computation interface). Further investigation is required to determine how efficient the implemented polynomial approximations are, but seeing as performance was not a critical factor in the requirements specification, the implemented polynomial approximation algorithms are sufficient in terms of computation speeds.

It should however be noted that all of the Bézier curve approximations are done entirely on the CPU. Due to time constraints associated with the provided development time frame for implementing the requirements specifications, investigation into GPU processing based 3D curve generation and approximation techniques was omitted. This is something to perhaps look into when implementing any requirements specification, concerning 3D (or 2D) Bézier curve generation, in the near future. The advancement of GPU processing features associated with the introduction and standardization of compute and geometry shaders allow for very flexible and sophisticated use of Bézier curve generation techniques.

Summary and Conclusion

The implement requirements specification for the provided Bézier curve coursework meets all of the original stated requirements. The only feature not (directly) added was that of user based curve control point manipulation during the run-time of the application. This was omitted as the slope gradient approximations are not dynamic and are pre-calculated. However, the default curve generation control point coordinates are not hard coded and are loaded from three separate text files in the *data* folder directory of the main application. Should the user wish to edit these coordinates, they can, but they will also have to adjust the hard-coded slope gradient values.

In conclusion, the development of this coursework application has highlighted the importance of using Bézier curves in a 2D/3D scene, as a viable scene entity motion-control solution.

References

Dempski, K. 2003. *Focus on Curves and Surfaces.* Premier Press.

Farin, G. 1995. *NURB Curves and Surfaces: from Projective Geometry to Practical Use.* A K Peters, Ltd.

Vince, J. 2006. *Mathematics for Computer Graphics.* Springer.

Slater et al. 2002. *Computer Graphics and Virtual Environments: From Realism to Real-Time.* Addison Wesley.

McConnell, J. J. 2006. *Computer Graphics: Theory into Practice.* Jones and Bartlett.

Miller, C. J. 1998. *Bézier Curve Drawing in OpenGL.* Available Online [http://cs1.bradley.edu/public/jcm/cs535BézierCurve.html]

Pipenbrinck, N. 1999. *De Casteljau's Algorithm.* Available Online [http://cubic.org/docs/Bézier.htm]

Neumann, E. 2004. *Roller Coaster Physics Simulation.* Available Online [http://www.myphysicslab.com/RollerSimple.html]

Humphrey, B. 2005. *OpenGL Bézier Curve.* Available Online [http://www.gametutorials.com/gtstore/pc-66-1-Bézier-curve.aspx]

Twigg, C. 2003. *Camera Movement Along a Spline.* Available Online [www.cs.cmu.edu/~462/projects/assn2/assn2/cameraMovement.pdf]