Carleton University

COMP 4905 - Honours Project

Using Bezier Curves to Create an Intuitive Track Editor for Racing Games

Author: Nathan Bell, 100734700

Supervisor: Dr. Doron Nussbaum, School of Computer Science

Date: April 16th 2012

Abstract

This project was done with three goals in mind. The first and foremost was to create a system which used Bezier curve methods to create smooth 3D track meshes for racing games. The second goal was to create an editing program which used this track mesh generation system to offer a simple and intuitive way of creating full race tracks with smooth, complex curves. The final goal of this project was to then provide a simple racing game which used the tracks created with the previous systems to determine the quality of the tracks in a hands on way.

Acknowledgments

I would also like to acknowledge Paul de Casteljau, creator of De Casteljau's Algorithm, which lies at the heart of my system.

Table of Contents

List of Figures

1 : Background

1.1 : Motivation

There exist level editors for many game engines, and editors for many genres. However, editors for racing games have always been quite limited. It is rare in the first place for a racing game to have an editor, and when they do they're usually quite inflexible, using a number of pre-created blocks that you piece together or some other similar system. This is likely due to the fact that it is extremely difficult for the average user to sculpt a smooth track mesh, a tedious and slow process. A single polygon out of place on a track surface can cause serious problems. What a good track editor needs is some way of allowing simple manipulation without limiting flexibility.

Bezier curves offer a simple way of creating complex lines and are widely used in graphical design software. With this project I hope to use Bezier curves to generate smooth 3D track meshes, allowing for a flexible, yet simple track editor.

1.2 : Choice of Language and Environment

For this project I chose to develop using the C# language and the Microsoft XNA Game Development environment.

The Microsoft XNA Environment provides a framework for game development, a wrapper for direct x and many useful tools and libraries. I chose to use XNA as I had experience working with it from my game development courses and it allowed me to focus more on developing the important game aspects, rather than being bogged down with building my own engine from the bottom up.

1.3 : Bezier Curves

Bezier curves are parametric curves which define a smooth path using a number of "control points". These control points can be vectors representing anything, but typically represent a point in space. A point on a Bezier curve is defined by a function of a value t between 0 and 1 with 0 representing the start of the curve and 1 representing the end. This function interpolates between n control points according to the following equation:

$$B(t) = \sum_{i=0}^{n} \binom{n}{i} (1-t)^{n-i} t^i P_i$$

Where P_i is control point i.

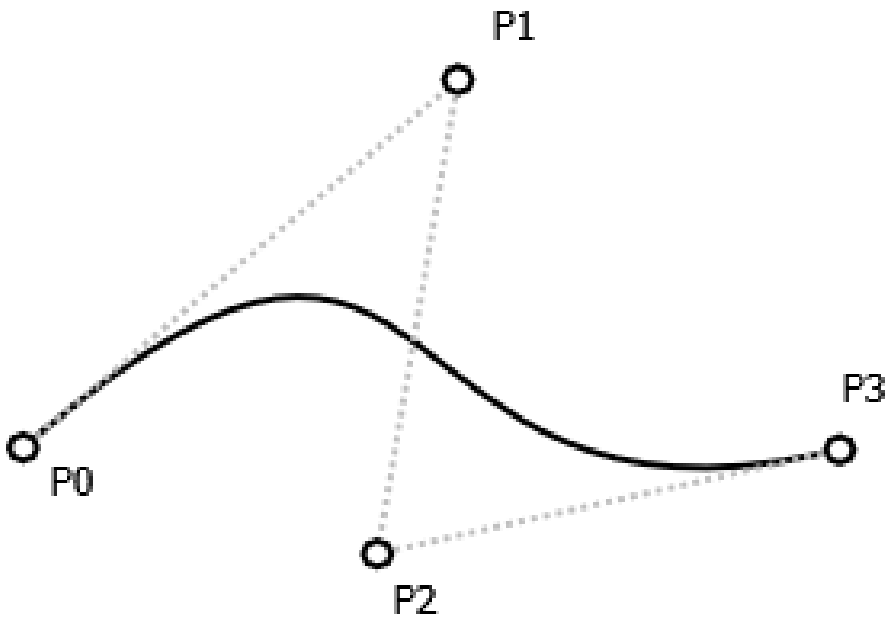


FIGURE 1 : A two dimensional Bezier Curve with four control points

Bezier curves are widely used in graphic design programs and for animation, due to the intuitive and visual nature of the control points.

1.4 : Existing Systems for Track Editors

Most track editing programs fall under the same style, which I will refer to as the "Puzzle" style of editor. In this style of editor, the user is given a library of pre-modeled track pieces which the user then lays down like puzzle pieces, connecting them together to create a track. These systems nearly always feature a grid of some sort which restricts the users placement choices further to simplify the placement of track pieces.

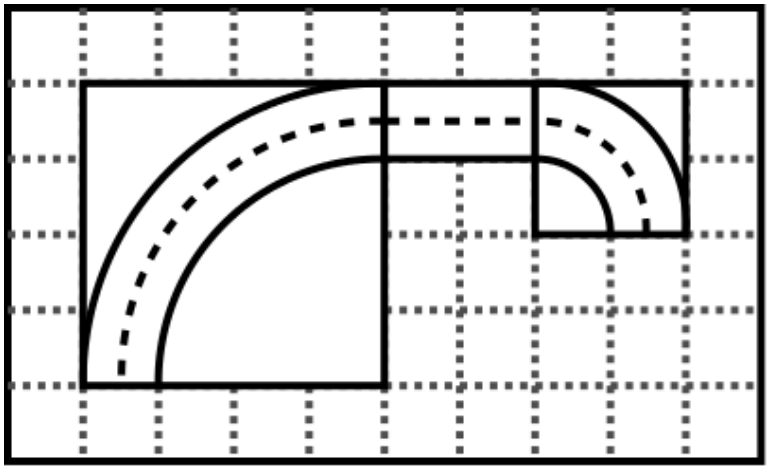


FIGURE 2 : Illustration of the Puzzle System

Systems like these can be found as early as 1990, with the game Stuntz for PC and Amiga and are still being used today by games such as the TrackMania franchise.

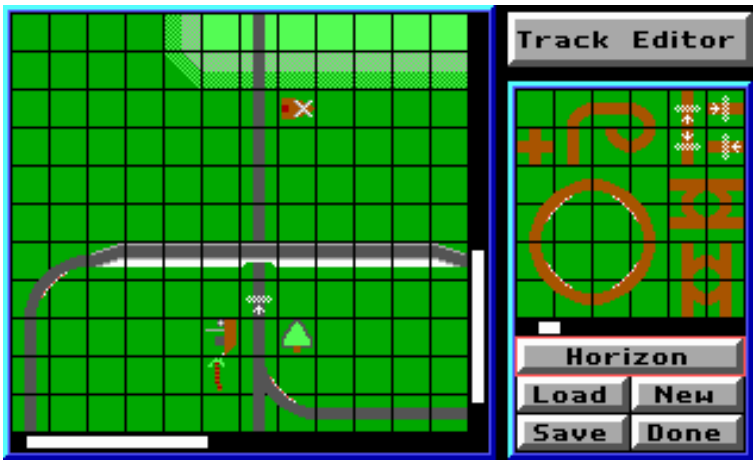


FIGURE 3 : Stunts, 1990

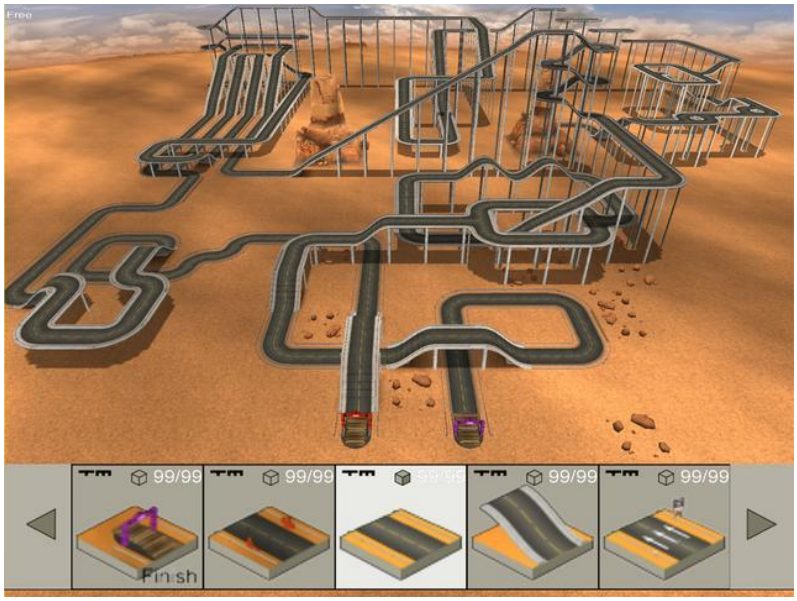


FIGURE 4 : TrackMania, 2003

Despite the advancement of the power and graphical ability of computers, the underlying system has not evolved much over the years. Apart from extending the system to be able to build in 3D and adding a wider variety of pieces, the tracks are still locked into the grid system and the flexibility of the editor can only go as far as the choice of pieces.

2 : Implementation

2.1 : Overview of System

In this system a track is a list of track points, which are used in the same was as control points are used in Bezier curves. If the whole track was to be treated as a single curve with many points, the result would be undesirable. This is because Bezier curves interpolate between points, and are only guaranteed to pass through the beginning and end points on their curve. Instead of this, the track points are divided up between a series of track segments. Each track segment is treated as a separate curve. If two track segments are connected and continuous, it must be ensured that the previous track segment ends with the same orientation as the next track segment. This is done by ensuring that the two points before and after the connecting point, and the connecting point itself are always collinear. The fact that the two continuous segments share a common connecting track point ensures that the roll and width will be continuous.

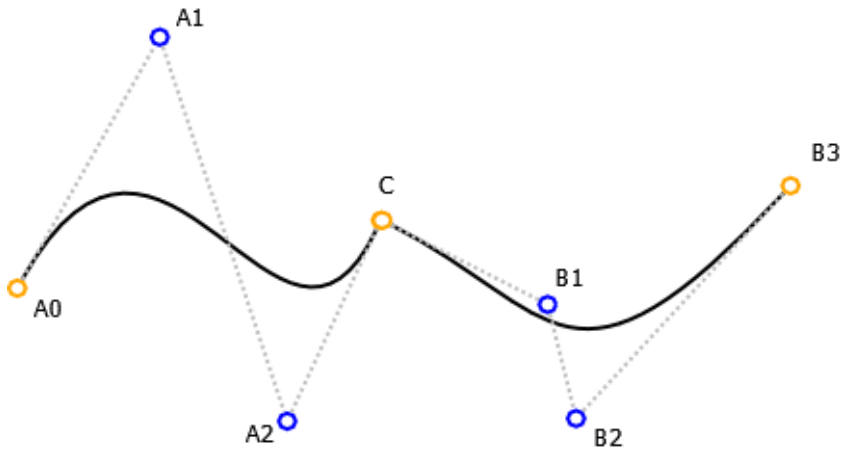


FIGURE 5 : Curves A and B sharing common point C. The two curves do not transition smoothly

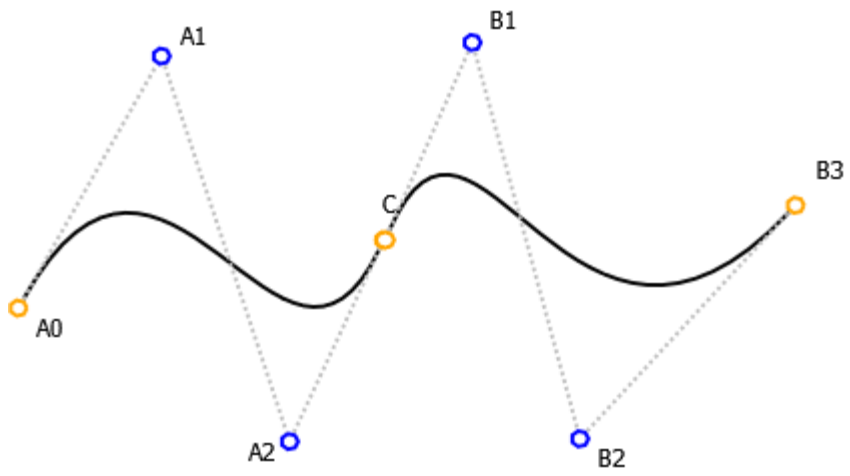


FIGURE 6 : Points A2, C and B1 are made to be co-linear, resulting in a smooth transition

The project is divided into two main sections, the track editor and the racing game. The track editor portion contains most of the original algorithms and ideas, as it is where all the track generation takes place.

Because of the flexibility of the Bezier curve system, I decided not to limit the tracks by making a realistic style of racing game. In the racing game, the player's vehicle "sticks" to the track surface, ignoring gravity unless it falls off the track. This allows the vehicle to traverse the track even with extremely complex inversions and curves. Collision detection for the vehicle is done using Sphere - Polygon collision techniques. The player vehicle has three collision spheres, one which is used to keep the car from flying off the track over hills, one to keep the car from passing through the track, and one used to detect collisions with the walls of the track. This collision detection is explained in greater detail in the Description of Important Algorithms section. The style of racing is time trial, after the first lap, the time at each checkpoint, and the relative time compared to the previous lap are displayed.

The following is a brief description of all the notable classes in this project.

2.2 : Track Editor Classes:

TrackEditor:

The TrackEditor class is a singleton pattern class holding a track object and all functions which modify the track in any way. This includes the algorithm which ensures that track segment connections remain collinear.

Track:

This class holds a list of Track Segments and Track Points, as well as a list of Editor Checkpoints. This class contains nothing else important but the draw call for the track in editor mode.

Track Point:

Track points contain the essential information used by the track generation algorithms. A point contains a position in 3D space, a roll and a width. The Track Point also contains pointers to the next and previous Track Point in the track

TrackSegment:

The TrackSegment class holds a list of Track Points and a mesh. This class contains the track mesh generation algorithm, the track mesh chunking algorithm and the modified Casteljau algorithm. These algorithms are discussed in detail in the Description of Important Algorithms section.

Track Cross Section:

A Track cross section represents a cross section, or slice, of the track mesh. This class contains two separate lists of vertices. The first is a list of vertices representing the portion of the cross section which will be stretched according to width. The second represents the walls and edge of the track segment. Instead of being stretched, the wall points are moved according to the width, to avoid the wall becoming warped.

PresentationPoint:

Presentation points are used to graphically represent track points in the editor view. Colours change depending on if the associated point is an endpoint, selected by the user, or neither of these.

2.3 : Racing Game Classes:

RacingGame:

The Racing Game class is a singleton pattern class holding all information and functions needed to run the automated portion of the game logic. Rather than directly use the Track Segment classes, the Racing Game instead holds a list of Track Chunks generated from the Track Segment classes. At each update, collisions are processed and the player's vehicle is updated. To deal with the fact that the collision detection is not predictive, collision processing and car updating occurs three times per update.

RaceView:

The Race View class extends Game View and handles all user interaction with the Racing Game class. The handleInput method allows control of the player car using the keyboard and the draw function displays a basic heads up display with information about the cars speed and lap times.

TrackChunk:

Generated by the Track Segment chunking algorithm, this class holds a 'chunk' of track mesh and a bounding sphere which contains all the vertices in the mesh. These objects are used rather than Track Segments to allow faster collision detection.

Car:

This class represents a vehicle. The update function moves the car automatically based on its velocity, heading, acceleration, and a number of other factors. The player's car is controlled via the Race View class by setting various flags for the update. The car class does not handle any of its own collision detection or resolution.

RaceCheckpoint:

This class simply represents a checkpoint along the track. This class stores a pointer to the next checkpoint and the current lap time when the car passes it. This time is used to judge roughly how well you are doing compared to previous laps.

2.4 : Other Classes:

GraphicsManager:

The graphics manager class is a singleton pattern class which stores and manages the information needed to draw to the screen. Content such as textures, models and other outside files are loaded using XNA's content system and stored in several dictionary data structures. In total, there are dictionaries storing Effect, RasterizerState, Texture, Model and Font objects. This system offers a centralized approach to storing this information, rather than all draw able classes containing direct references and requiring content loads themselves. This also allows for some small optimizations, as the centralized graphics manager knows what is currently loaded on the device, and can avoid redundant calls.

View:

The view class holds everything needed to calculate the projection and view matrices needed for displaying the 3D world to the user. Views hold a Camera, a Viewport and a string representing a rasterizer state. Viewports are an XNA class representing the dimensions of an output window or screen. View objects also contain a method which translates a mouse click on screen into a ray in the game world using XNA's Viewport unproject function, which is essential for mouse based input.

Camera:

The camera class and its subclasses represent various types of cameras. Cameras of all types store a view matrix which represents the cameras orientation in space and what it 'sees'. Different subclasses of the camera class vary in how their position and rotation are determined, making different camera types better for different situations.

FileManager:

This class holds all methods used for saving and loading tracks from file. The file format used to store tracks on file is fairly simple. First, a start of file line is written, followed by the number of track points, segments and checkpoints in the track. Next, a line stating the beginning of the track point section is written, followed by writing each point to file. Each point is written to file in the following way. First the X, Y and Z values of the points position are written, then the roll and width values. "true" or "false" is then written depending on if the point has a pointer to a previous point and then finally the same is done for a pointer to the next point. After this, a line stating the beginning of the track segment section is written. For each track segment, "true" or "false" is written depending on if the segment has a previous connecting segment. This is followed by the order of the segment. Finally a "true" of "false" is written depending on if the segment has a next connecting segment. A line stating the beginning of the checkpoint section is written. For each checkpoint, the index of the checkpoints associated track point is written to file. Finally, the end of file line is written and the file is closed.

Tracks are loaded from file by reading all this information in and recreating the track from it.

3 : Description of Important Algorithms

3.1 : Modified De Casteljau's Algorithm

De Casteljau's Algorithm is a recursive algorithm for calculating Bezier curves of arbitrary order. Given n control points and a value t (between 0 and 1), De Casteljau's Algorithm will return the point along the Bezier curve corresponding to the value t.

```
Point CasteljauPoint( int r, int i, float t)

    if( r == 0 ) return points[i]

    Point a = CasteljauPoint ( r - 1, i, t )
    Point b = CasteljauPoint ( r - 1, i + 1, t )

    return  ( ( 1 - t ) * a ) + ( t * b )
```

Extending this algorithm to 3D space is just a matter of using three dimensional vectors rather than two dimensional points. For the purposes of track generation, getting the position along the curve is not enough. Not only did I require a 3D point, I needed slope, roll and track width. Fortunately, De Casteljau's Algorithm is quite flexible, and having it interpolate these values as well was easy. Roll and width could simply be calculated by adding them as additional dimensions to the vector, and calculating slope required only minor additional calculations. Due to XNA's built in Vector classes not supporting arbitrary dimensions, I decided it would be easier to handle an array of three 3D vectors, each holding part of the information.

```
Vector3[] CasteljauPoint( int r , int i , float t )

    Vector3[] result = new Vector3[ 3 ]

    if ( r == 0 )
        result[0] = points[i].Position
        result[2].X = points[i].Roll
        result[2].Y = points[i].Width
        return result

    Vector3[] a = CasteljauPoint( r - 1 , i , t )
    Vector3[] b = CasteljauPoint( r - 1 , i + 1 , t )

    result[0] = ( ( 1 - t ) * a[0] ) + ( t * b[0] )
    result[1] = Normalize( b[0] - a[0] )
    result[2] = ( ( 1 - t ) * a[2] ) + ( t * b[2] )

    return result
```

Where result[0] is the position in 3D space, result[1] is the slope as a unit vector and result[2] holds roll and width information.

This algorithm runs $O(n^2)$ where n is the number of control points.

3.2 : Track Generation Algorithm

Using information from the modified De Casteljau's Algorithm, this algorithm constructs the full mesh of the track segment.

Using a list of track points, a Bezier curve is defined. Using the modified Casteljau algorithm, position, orientation, roll and width information is calculated for a number of points along the line. The number of points desired is pre-calculated and is determined by a rough approximation of the curve length and a general desired resolution. For each of these points of information, a track cross section object is copied and transformed. The cross section is rotated according to its orientation and roll values and transformed according to its position and width values. Vertices are then generated from this cross section, added to the track segment mesh and indices generated to add the desired polygons.

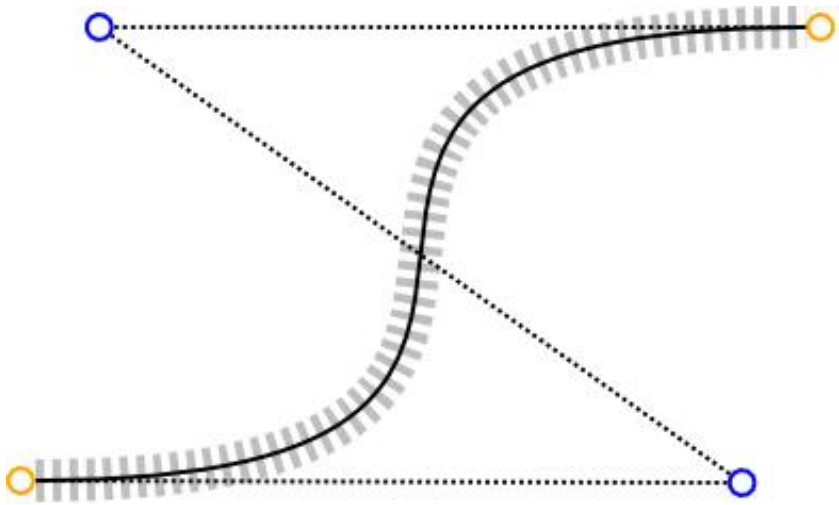


FIGURE 7 : A Bezier curve with shaded lines indicating position and orientation of the segments cross sections

Originally I planned on directly calculating transformation and rotation matrices for each point along the curve, but ran into some serious issues with that method. First and foremost was the issue of gimbal lock. This was a serious issue, as the track must bend as smoothly and fluidly as possible. Generating track which passed through a point with a vertical slope would cause the track to instantly flip orientations, breaking the track surface entirely. To address this I decided to switch to using Quaternions for rotation. Quaternions are not prone to gimbal lock, solving the issue with vertical slopes.

However, the quaternion system raised a severe problem of its own. Quaternions represent an axis and a rotation about said axis. The rotation, however, cannot exceed 180 degrees. This caused a big issue when the track segment curved more than 180 degrees, which is a very common occurrence. At this point, rotations were being calculated by spherically interpolating between a beginning and ending rotation quaternion. The spherical interpolation algorithm would calculate an intermediate quaternion and would always take the "shortest path" from the beginning to end quaternion. This meant that if the ending quaternion was over 180 degrees different than the starting quaternion, the spherical interpolation function would cause the track to bend the wrong way, resulting in a track which folded in on itself.

It was clear at this point that dramatic changes had to be made in the way I calculated the rotation along the curve. My solution was to adopt a method of "nudging" the quaternion as I calculated information for each point along the line. In order to do this, at each point I would calculate a delta rotation using the slope of the current point and slope of the previous point. A temporary Quaternion representing this change in rotation would be created. The axis of rotation would be obtained from the cross product of each slope vector, and the rotation value itself would be obtained using the dot product of these two vectors.

At the beginning of the segment, a rotation quaternion would be created, representing the starting rotation of the curve. For each point along the curve, the temporary delta rotation quaternion would be applied to the current rotation quaternion, "nudging" it to the correct orientation. The difference between two consecutive points on the curve will never be greater than 180 degrees unless desired, so this method successfully calculates the proper orientations at each point to create a smooth curve.

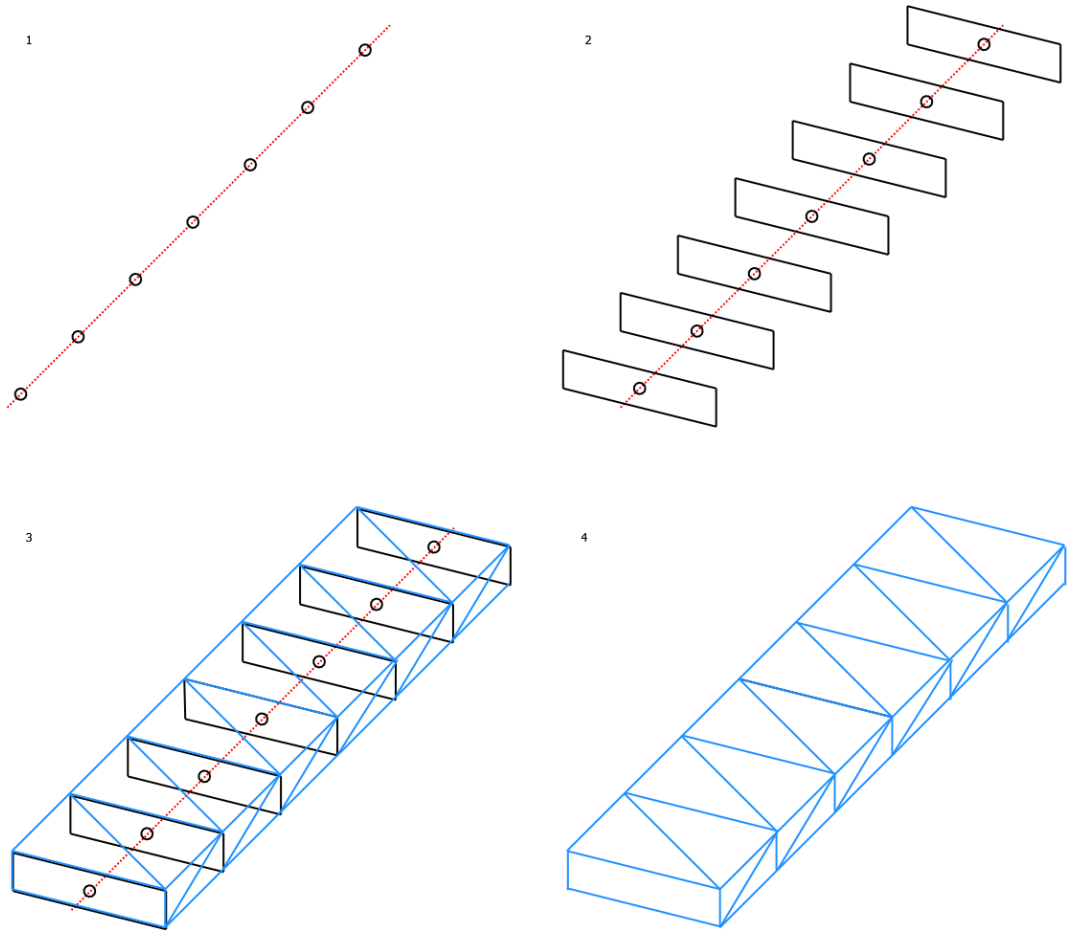


FIGURE 8 : An Illustration of the Mesh Generation Process

The final result of this algorithm is a fully generated mesh object for this track segment. Because these control points can be in any orientation, the lengths of different segments can vary to extremes. To prevent long curves having too few points, or short curves having too many, the length of the curve is roughly estimated beforehand, determining the number of points calculated.

```
Quaternion tempRotationQuaternion
Quaternion tempDeltaQuaternion
Quaternion tempRollQuaternion
Vector3 tempRotationAxis
Vector3[] point
Vector3[] lastPoint
List<VertexPositionNormalTexture> tempVert
CrossSection crossSection
float t
float rotation
For( index = 0; index <= resolution; ++index)

        t  = index / resolution
        if( index > 0 )
                lastPoint = point
                point = CasteljauPoint ( Order - 1, 0, t )
        else
                point = CasteljauPoint ( Order - 1, 0, t )
                lastPoint = new Vector3[3]
                lastPoint[0] = position of point
                lastPoint[1] = Vector3.Backward
                lastPoint[2] = roll and width vector of point

        tempRotationAxis = Normalize( CrossProduct( lastPoint[1], point[1] ) )
        rotation = ArcCos ( DotProduct ( lastPoint[1], point[1] ) )
        tempDeltaQuaternion =
                Quaternion.CreateFromAxisAngle (tempRotationAxis, rotation)

        tempRotationQuaternion = tempDeltaQuaternion * tempRotationQuaternion
        tempRollQuaternion =
                Quaternion.CreateFromAxisAngle (point[1], point[2].X )
                 * tempRotationQuaternion

        crossSection = new CrossSection ( point[0], tempRollQuaternion )
        tempVert.Add ( crossSection.Vertices )

Calculate indices and normals for the mesh
```

This algorithm runs in $O(m * n^2 + m * s)$ where m is the number of points along the segment to be evaluated, n is the number of control points of the segment and s is the number of vertices in the track cross section being used.

3.3 : Track Mesh 'Chunking' Algorithm

This algorithm is used when converting a track in the editor to a final, race-able track mesh. Because of the wildly varying lengths of the track segments, it would be impractical to use those meshes for the racing game aspect. Instead we want to chop the track up into more manageable, and more uniformly sized, chunks. This algorithm does just that. It works by dividing the mesh generated by the calculate vertices algorithm into sections which fit well into bounding spheres. The fact that they fit well into bounding spheres is important for collision detection when racing.

```
List<TrackChunk> chunks
VertexPositionNormalTexture[] vertices
int[] indices
numVPerSlice = number of vertices in cross section
int resolution = number of cross sections used in the track mesh
int numberOfSlices
int vertIndex = 0
int sInd
float dist
int i = 0

While ( i < resolution - 1 )
        dist = 0
        numberOfSlices = 0

        While ( i + numberOfSlices < resolution && dist < widths[i] )
                dist += distance between point i and point i + 1
                ++numberOfSlices

        if  ( i + numberOfSlices == resolution - 1 )
                ++numberOfSlices

        sInd = 0
        vertices = new VertexPositionNormalTexture[ (numberOfSlices + 1) *
                numVPerSlice ]
```

```
While ( sInd <= numberOfSlices )
        For ( int j = 0 ; j < numVPerSlice ; ++j )
                vertices[ (sInd * numVPerSlice ) + j ] =
                        MeshVertices[ vertIndex]
                ++vertIndex
        if ( sInd < numberOfSlices )
                Set up indices of new mesh
        ++sInd

vertIndex -= numVPerSlice
chucks.Add( new TrackChunk ( vertices, indices, TextureOfSegment ) )
i += numberOfSlices
```

3.4 : Sphere-Track Chunk Collision Detection

When racing on a track of any reasonable size it would be infeasible to do hit detection on every single polygon at every frame. To address this we need some way of quickly narrowing the search to a manageable number of polygons. When racing, the track is stored in a list of Track Chunk objects. Track Chunk objects have a pre-calculated bounding sphere which surrounds all vertices in the chunk. Before running any collision detection on individual polygons, we first check collisions with these bounding spheres. If there is no collision with the sphere, it is impossible that there is a collision with any polygon in that chunk, so we can ignore it. After running this preliminary collision detection we can narrow down the search to, on average, a single track chunk. This means that collisions can be processed for very long tracks without impacting performance.

Polygons to be tested for collision are loaded into three arrays, each array representing a corner of the polygon. Each polygon is then tested iteratively against a single collision sphere. This iterative approach is more efficient than calling a function for each individual polygon.

For each polygon, a 3D plane is defined. A ray is then defined using the negated plane normal and the position of the collision sphere. Ray - Plane intersection is performed to find the point on the plane closest to the position of sphere. If the distance between the sphere and this point is greater than the radius of the sphere, there is no collision and we move on to the next polygon. If the distance is less than the radius of the sphere, we must test whether or not the point on the plane is within the polygon.

To test whether or not the with the plane lies within the polygon I calculate a number of angles using dot products and compare four angles to determine whether or not there is a collision.

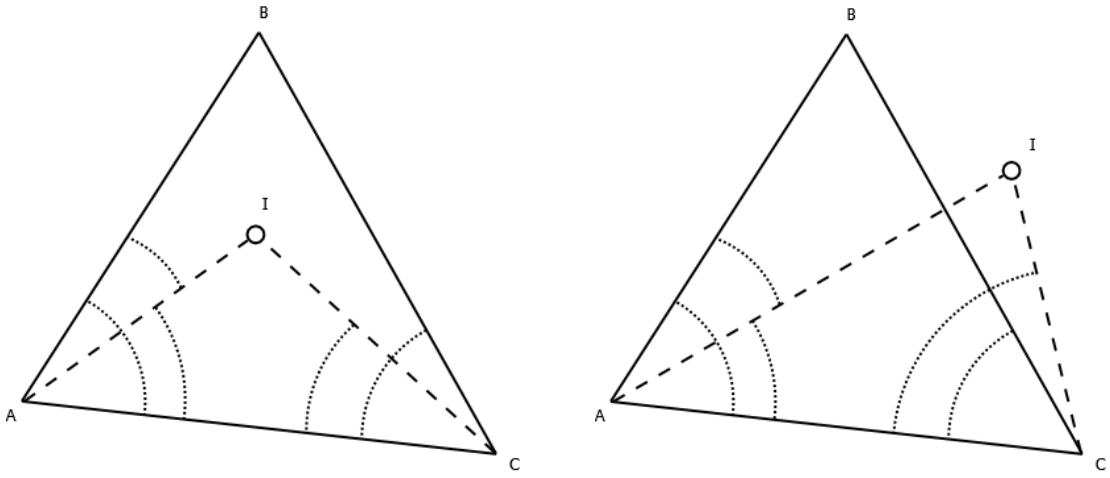


FIGURE 9 : Illustration of Method to Determine Whether or not a Point Lies Within the Polygon

First, angles BAI and IAC are compared to angle BAC. If the angles BAI and IAC are both less than BAC a collision is possible. We must then determine on what side of BC the point of intersection lies. I do this by comparing angle ICA to angle BCA. If ICA is less than BCA, the point of intersection lies within the polygon and a collision has occurred.

If a collision is detected then the point of intersection is added to a list, which is returned after all polygons have been tested.

```
Vector3[] a = list of 'a' points of the polygons to be tested
Vector3[] b = list of 'b' points of the polygons to be tested
Vector3[] c = list of 'c' points of the polygons to be tested
Vector3[] normal = list of normals of the polygons to be tested
List<Vector3> collisions
Plane p
Ray r
Vector3 pointOfIntersection
Vector3 ab
Vector3 ac
Vector3 ai
Vector3 bc
Vector3 bi
double thetaA
double thetaB

For ( int i = 0 ; i < number of polygons ; ++i )
        p = plane defined by a[i], b[i] and c[i]
        r = ray defined by the position of the collision sphere and the inverse of normal[i]
        len = distance from the position of the collision sphere to the point of intersection
with the plane p
        if ( len > 0 && len < radius of collision sphere )
                pointOfIntersection = point of intersection on plane p
                ab = Normalize ( b[i] - a[i] )
                ac = Normalize ( c[i] - a[i] )
                ai = Normalize ( pointOfIntersection - a[i] )
                bc = Normalize ( c[i] - b[i] )
                bi = Normalize ( pointOfIntersection - b[i] )
                thetaA = Arccos ( Dot product of ( ab, ac ) )
                thetaB = Arccos ( Dot product of ( -ab, bc ) )

                if ( Arccos ( Dot product of ( ab, ai ) ) < thetaA
                        AND Arccos ( Dot product of ( ai, ac ) ) < thetaA
                        AND Arccos ( Dot product of ( bi, bc ) ) < thetaB
                        AND Arccos ( Dot product of ( -ab, bi ) ) < thetaB )

                        add pointOfIntersection to collisions

        return collisions
```

4 : Conclusions

4.1 : Appraisal of Final Product

All three goals of this project have been achieved. The track editor is quite easy to use, and the Bezier curve system turned out to be as intuitive as I hypothesized. It is clear how moving the control points affects the curves, so the process of creating a track is quite visual and makes sense without needing knowledge of the underlying process.

Track mesh is generated pretty much perfectly from the Bezier curves, the surface is smooth and consistent throughout the track. Below are some screenshots of the generated meshes in editor mode.

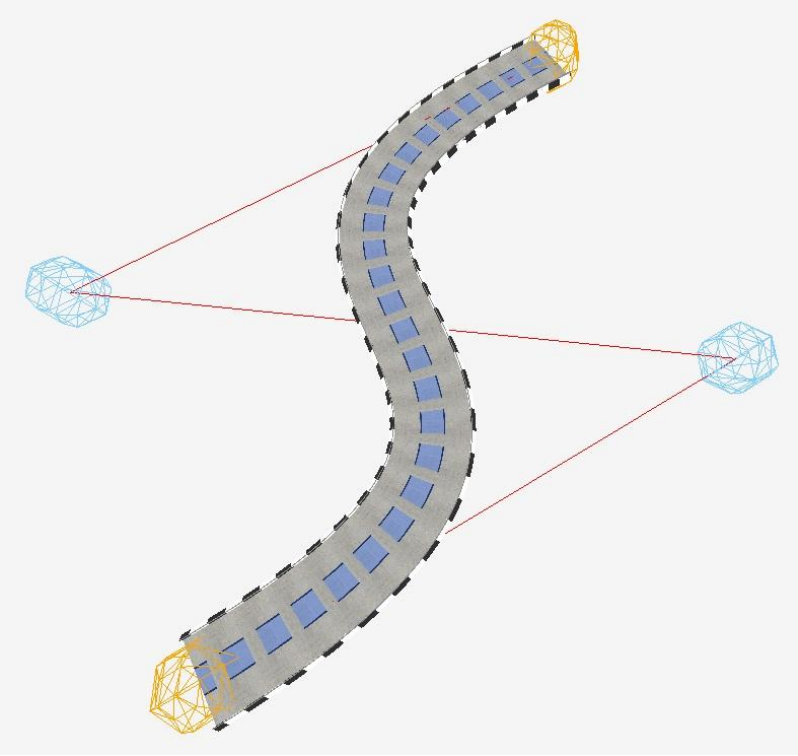


FIGURE 10 : A Simple Track Segment

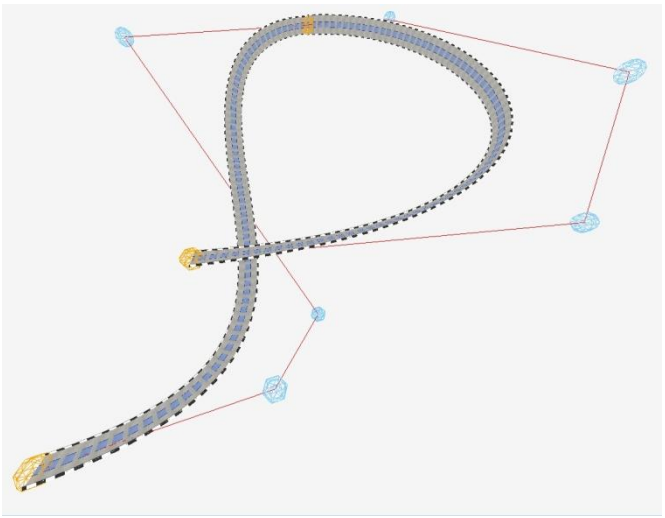


FIGURE 11 : Two Connected Segments Creating a Slightly more Complex Shape

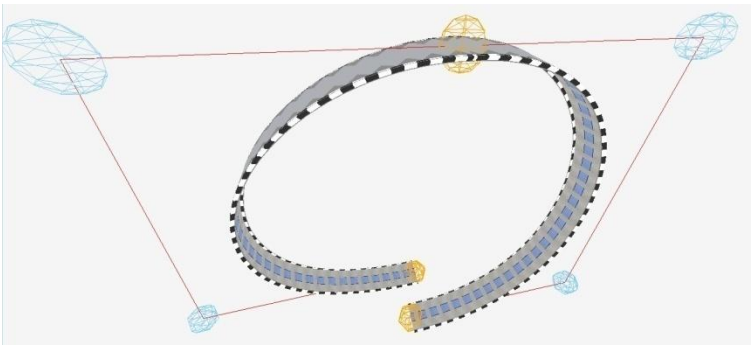


FIGURE 12 : A Classic Loop Created With Two Segments

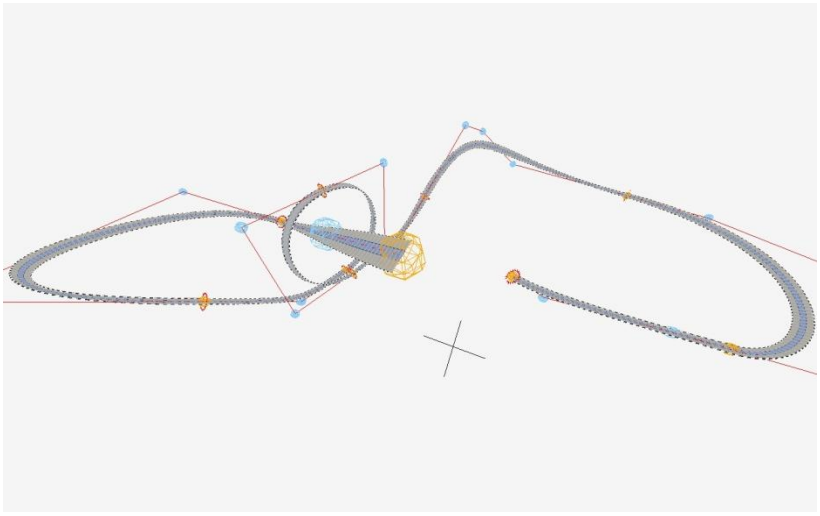


FIGURE 13 : A Full Track

The racing game aspect ended up working as well as I expected. By running collision detection multiple times per frame I was able to work around the limitation of frame dependant detection without limiting the speed. It is still possible however to clip through the track at extreme speeds. Aside from this and a few minor glitches, the car travels smoothly along the track as you would expect.

There are a couple errors in the implementation of the algorithm that were unresolved before evaluating the algorithm. In some cases, when generating the mesh, a small error in the roll of the track would crop up. To solve this, I have the error propagate to the next segment to compensate. Because of this, in these cases it may appear as though the track becomes discontinuous when this error changes. Moving a point in the next segment to manually cause an update on that segment will fix the issue. I decided not to have this happen automatically, as it could end up causing a drastic decrease in performance, as in some cases all segments would have to be recalculated.

Currently, the track must be a single continuous path, but cannot be a full cycle. This is the first thing I would address, had I more time. This issue stems from some unexpected behaviour when tracks have multiple, independent cycles. I believe this problem stems from the roll error discussed above.

As mentioned above, the player's vehicle is capable of clipping through the track when traveling at extreme speeds. Additionally, a similar issue occurs when the vehicle travels over a hill at high speeds and fails to "stick" to the track.

4.2 : Possible Improvements

There are many extensions to this project which I would like to have done. The Bezier interpolation could be easily extended to interpolate many more things, such as track textures and cross sections. This would allow fluid transitions between different kinds of track surfaces, adding even more flexibility to the editor. For this interpolation to work properly, track cross sections would need a consistent number of vertices. Positions of corresponding vertices would be interpolated to create the intermediary cross sections. Providing a way for the user to create custom cross sections would also work nicely with this.

The collision detection during racing could also be improved by doing predictive detection using the vehicles velocity vector, rather than moving and checking every step. This would resolve some issues with the vehicle falling through the track at very high speeds. I did not implement it this way as this was the first time I've done any sort of 3D collision detection, so I wanted to keep it simple.

From a software engineering standpoint as well, there are a number of things I would change if starting over. I would use an event based system and totally rewrite the way the track is modified inside the TrackEditor class to make it more modular. I would do this by creating custom event handlers and command classes, rather than having functions in the TrackEditor class doing operations.

4.3 : Final Conclusions

Overall, the project was a success. After a couple rewrites and different approaches to implementing the track generation algorithm, track generation worked great. It's quite easy to create interesting tracks in under five minutes, which is the feeling I was going for with this system.

I believe that if polished and perfected, this sort of system could do a lot for reinvigorating racing games as a genre. If implemented in commercial game, this system could provide nearly limitless user generated content and would be a step forward from the puzzle systems of most track editors.

5 : References

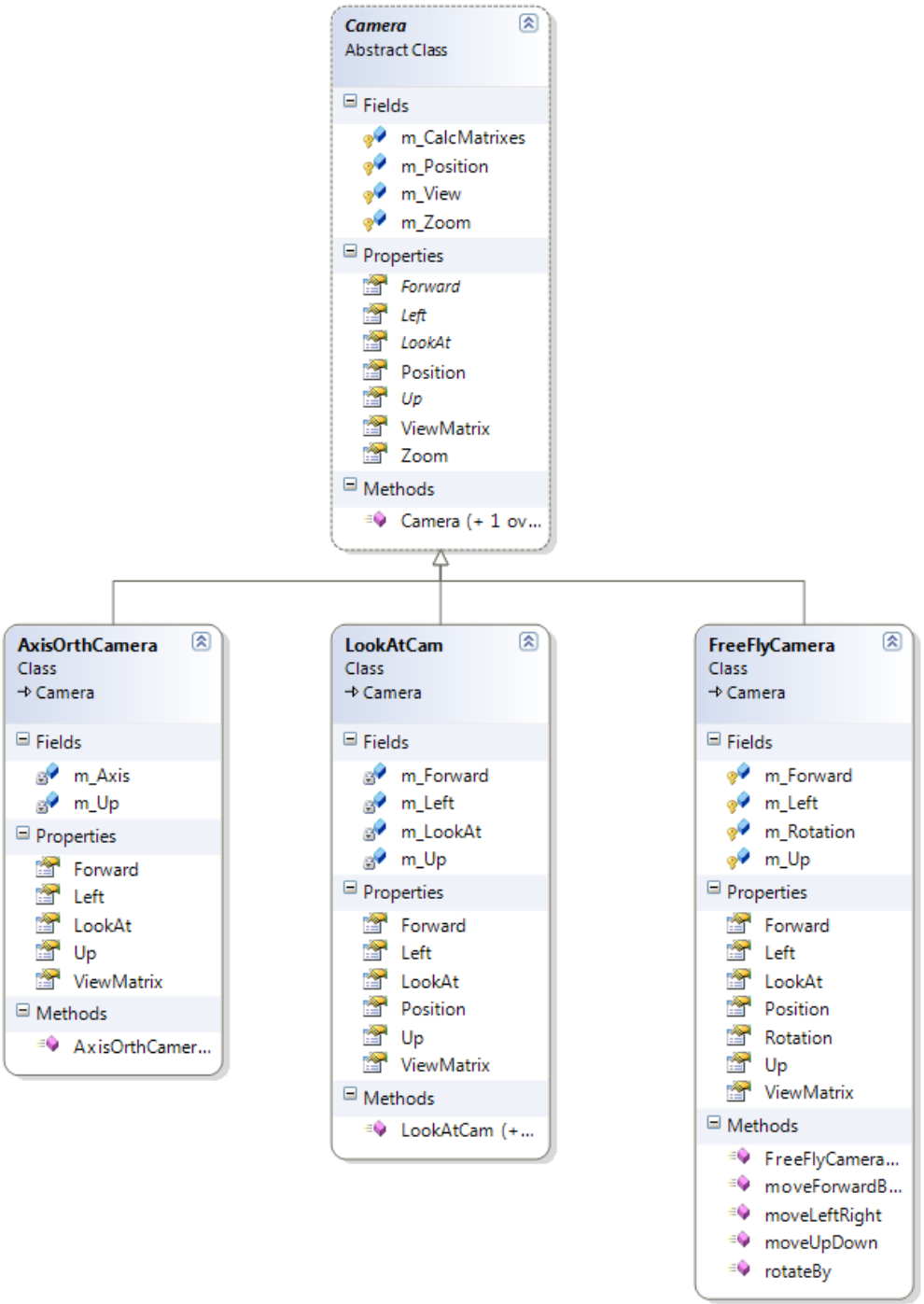
T. Sederberg, Bézier Curves,

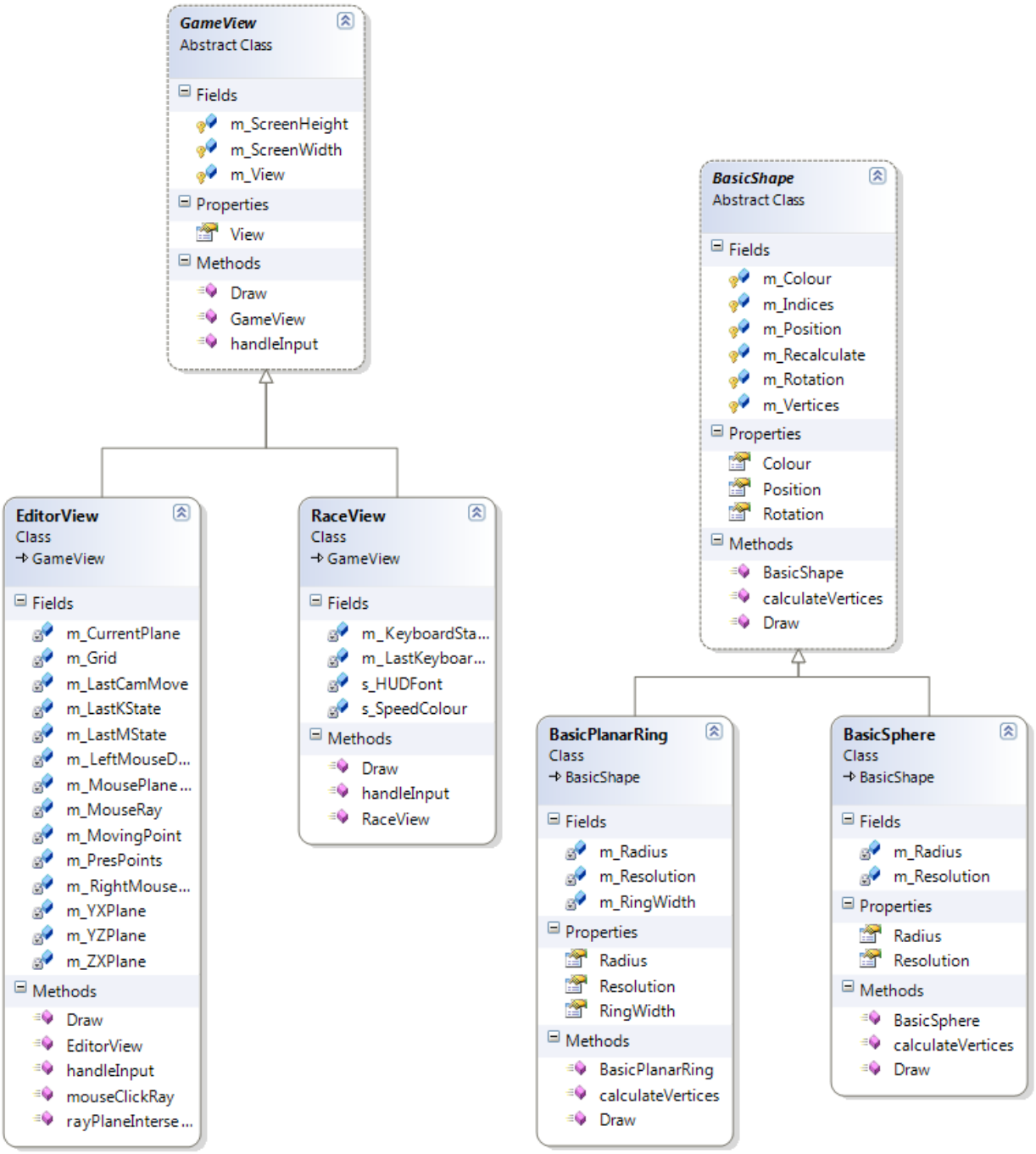
Internet: http://www.tsplines.com/resources/class_notes/Bezier_curves.pdf

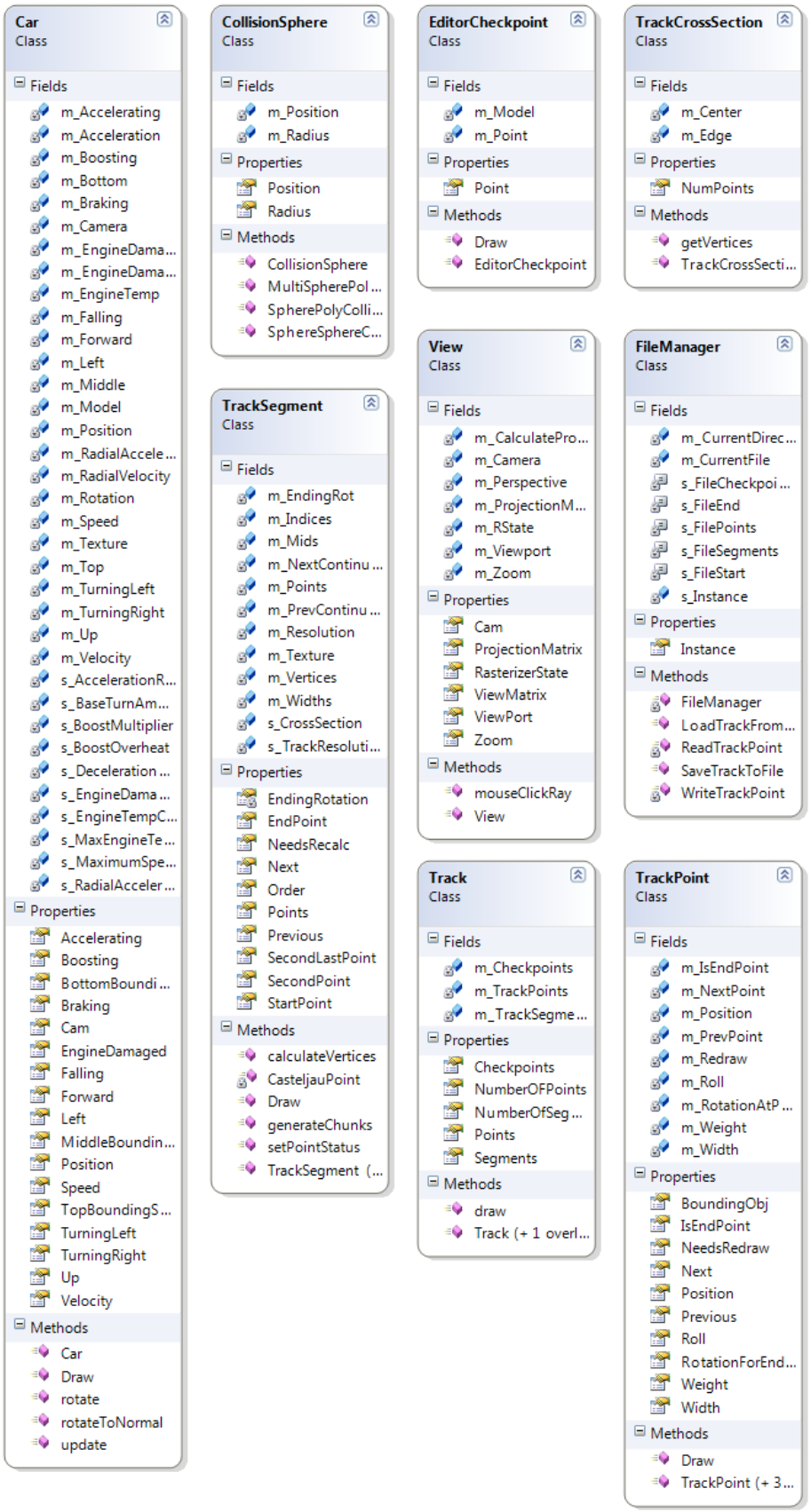
Moby Games, "Stunts", Internet: www.mobygames.com/game/dos/stunts, 1990

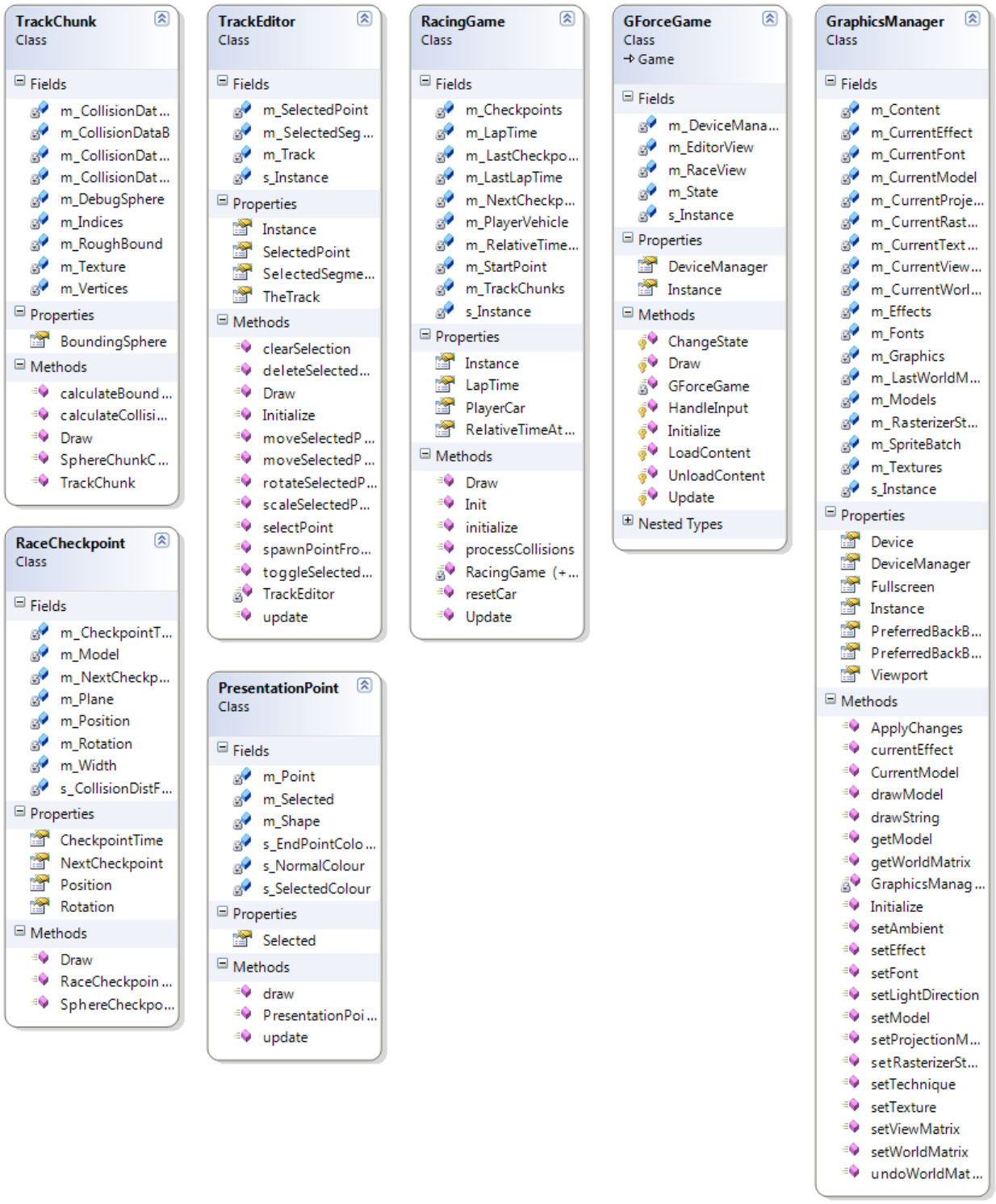
Nadeo, "TrackMania", Internet: http://official.trackmania.com/indexUk.php, 2005

Appendix A : Class Diagrams




GameView
Abstract Class
Fields
m_ScreenHeight
m_ScreenWidth
m_View
Properties
View
Methods
Draw
GameView
handleInput
BasicShape
Abstract Class
Fields
m_Colour
m_Indices
m_Position
m_Recalculate
m_Rotation
m_Vertices
Properties
Colour
Position
Rotation
Methods
BasicShape
calculateVertices
Draw
EditorView
Class
GameView
Fields
m_CurrentPlane
m_Grid
m_LastCamMove
m_LastKState
m_LastMState
m_LeftMouseD...
m_MousePlane...
m_MouseRay
m_MovingPoint
m_PresPoints
m_RightMouse...
m_YXPlane
m_YZPlane
m_ZXPlane
Methods
Draw
EditorView
handleInput
mouseClickRay
rayPlaneInterse...
RaceView
Class
GameView
Fields
m_KeyboardSta...
m_LastKeyboar...
s_HUDFont
s_SpeedColour
Methods
Draw
handleInput
RaceView
BasicPlanarRing
Class
BasicShape
Fields
m_Radius
m_Resolution
m_RingWidth
Properties
Radius
Resolution
RingWidth
Methods
BasicPlanarRing
calculateVertices
Draw
BasicSphere
Class
BasicShape
Fields
m_Radius
m_Resolution
Properties
Radius
Resolution
Methods
BasicSphere
calculateVertices
Draw


Car
Class
Fields
m_Accelerating
m_Acceleration
m_Boosting
m_Bottom
m_Braking
m_Camera
m_EngineDama...
m_EngineDama...
m_EngineTemp
m_Falling
m_Forward
m_Left
m_Middle
m_Model
m_Position
m_RadialAccele...
m_RadialVelocity
m_Rotation
m_Speed
m_Texture
m_Top
m_TurningLeft
m_TurningRight
m_Up
m_Velocity
s_AccelerationR...
s_BaseTurnAm...
s_BoostMultiplier
s_BoostOverheat
s_Deceleration...
s_EngineDama...
s_EngineTempC...
s_MaxEngineTe...
s_MaximumSpe...
s_RadialAcceler...
Properties
Accelerating
Boosting
BottomBoundi...
Braking
Cam
EngineDamaged
Falling
Forward
Left
MiddleBoundin...
Position
Speed
TopBoundingS...
TurningLeft
TurningRight
Up
Velocity
Methods
Car
Draw
rotate
rotateToNormal
update
CollisionSphere
Class
Fields
m_Position
m_Radius
Properties
Position
Radius
Methods
CollisionSphere
MultiSpherePol...
SpherePolyColli...
SphereSphereC...
TrackSegment
Class
Fields
m_EndingRot
m_Indices
m_Mids
m_NextContinu...
m_Points
m_PrevContinu...
m_Resolution
m_Texture
m_Vertices
m_Widths
s_CrossSection
s_TrackResoluti...
Properties
EndingRotation
EndPoint
NeedsRecalc
Next
Order
Points
Previous
SecondLastPoint
SecondPoint
StartPoint
Methods
calculateVertices
CasteljauPoint
Draw
generateChunks
setPointStatus
TrackSegment (...
EditorCheckpoint
Class
Fields
m_Model
m_Point
Properties
Point
Methods
Draw
EditorCheckpoint
View
Class
Fields
m_CalculatePro...
m_Camera
m_Perspective
m_ProjectionM...
m_RState
m_Viewport
m_Zoom
Properties
Cam
ProjectionMatrix
RasterizerState
ViewMatrix
ViewPort
Zoom
Methods
mouseClickRay
View
Track
Class
Fields
m_Checkpoints
m_TrackPoints
m_TrackSegme...
Properties
Checkpoints
NumberOFPoints
NumberOfSeg...
Points
Segments
Methods
draw
Track (+ 1 overl...
TrackCrossSection
Class
Fields
m_Center
m_Edge
Properties
NumPoints
Methods
getVertices
TrackCrossSecti...
FileManager
Class
Fields
m_CurrentDirec...
m_CurrentFile
s_FileCheckpoi...
s_FileEnd
s_FilePoints
s_FileSegments
s_FileStart
s_Instance
Properties
Instance
Methods
FileManager
LoadTrackFrom...
ReadTrackPoint
SaveTrackToFile
WriteTrackPoint
TrackPoint
Class
Fields
m_IsEndPoint
m_NextPoint
m_Position
m_PrevPoint
m_Redraw
m_Roll
m_RotationAtP...
m_Weight
m_Width
Properties
BoundingObj
IsEndPoint
NeedsRedraw
Next
Position
Previous
Roll
RotationForEnd...
Weight
Width
Methods
Draw
TrackPoint (+ 3...


TrackChunk
Class
Fields
m_CollisionDat...
m_CollisionDataB
m_CollisionDat...
m_CollisionDat...
m_DebugSphere
m_Indices
m_RoughBound
m_Texture
m_Vertices
Properties
BoundingSphere
Methods
calculateBound...
calculateCollisi...
Draw
SphereChunkC...
TrackChunk
RaceCheckpoint
Class
Fields
m_CheckpointT...
m_Model
m_NextCheckp...
m_Plane
m_Position
m_Rotation
m_Width
s_CollisionDistF...
Properties
CheckpointTime
NextCheckpoint
Position
Rotation
Methods
Draw
RaceCheckpoin...
SphereCheckpo...
TrackEditor
Class
Fields
m_SelectedPoint
m_SelectedSeg...
m_Track
s_Instance
Properties
Instance
SelectedPoint
SelectedSegme...
TheTrack
Methods
clearSelection
deleteSelected...
Draw
Initialize
moveSelectedP...
moveSelectedP...
rotateSelectedP...
scaleSelectedP...
selectPoint
spawnPointFro...
toggleSelected...
TrackEditor
update
PresentationPoint
Class
Fields
m_Point
m_Selected
m_Shape
s_EndPointColo...
s_NormalColour
s_SelectedColour
Properties
Selected
Methods
draw
PresentationPoi...
update
RacingGame
Class
Fields
m_Checkpoints
m_LapTime
m_LastCheckpo...
m_LastLapTime
m_NextCheckp...
m_PlayerVehicle
m_RelativeTime...
m_StartPoint
m_TrackChunks
s_Instance
Properties
Instance
LapTime
PlayerCar
RelativeTimeAt...
Methods
Draw
Init
initialize
processCollisions
RacingGame (+...
resetCar
Update
GForceGame
Class
Game
Fields
m_DeviceMana...
m_EditorView
m_RaceView
m_State
s_Instance
Properties
DeviceManager
Instance
Methods
ChangeState
Draw
GForceGame
HandleInput
Initialize
LoadContent
UnloadContent
Update
Nested Types
GraphicsManager
Class
Fields
m_Content
m_CurrentEffect
m_CurrentFont
m_CurrentModel
m_CurrentProje...
m_CurrentRast...
m_CurrentText...
m_CurrentView...
m_CurrentWorl...
m_Effects
m_Fonts
m_Graphics
m_LastWorldM...
m_Models
m_RasterizerSt...
m_SpriteBatch
m_Textures
s_Instance
Properties
Device
DeviceManager
Fullscreen
Instance
PreferredBackB...
PreferredBackB...
Viewport
Methods
ApplyChanges
currentEffect
CurrentModel
drawModel
drawString
getModel
getWorldMatrix
GraphicsManag...
Initialize
setAmbient
setEffect
setFont
setLightDirection
setModel
setProjectionM...
setRasterizerSt...
setTechnique
setTexture
setViewMatrix
setWorldMatrix
undoWorldMat...

Appendix B : User Manual

Editor Mode:

F2: Race with Current track

Moving The Camera:

- To move the camera, click and drag on open space
- To rotate the camera, click and drag with the right mouse button
- To move the camera up and down, hold shift while dragging

Moving Control Points:

- Click and Drag a control point to move it along the XZ plane
- Hold shift to move the control point up and down along the Y axis

Changing the Roll of a Control Point:

- Click on a control point to select it
- Press A and D to change the roll of the point
- Hold Control to increase the rate at which the roll changes

Changing the Width of a Control Point:

- Click on a control point to select it
- Press W and S to change the width of the point
- Hold Control to increase the rate at which the width changes

Adding a control point to an existing segment:

- Hold down E
- Click on the control point you wish to insert the new point after, and then drag the new point out.

Adding a track segment:

- Follow the directions for adding a control point, but select the end point of a segment at the end of the current track and a new segment will be made with the new point.
- You cannot create two order two segments in a row, in this case a segment will not be created.

Deleting a Control Point:

- Select a control point by clicking on it.
- Press Delete to remove the point
- If a point is deleted from an order two segment, the segment will also be removed

Save a track to file:

- Press the F5 key while in editor mode
- Navigate to the desired save location
- Enter a name for the track
- Save the file

Load a track from file:

- Press the F6 key while in editor mode
- Navigate to the desired .trk file with the file explorer
- Open the .trk file

Racing Mode:

Accelerate:

- W

Decelerate:

- S

Turn Left:

- A

Turn Right:

- D

Boost:

- Shift

Reset car to last checkpoint:

- R

Appendix C : Running Instructions

I have included both the source code, and compiled program. In order to compile this project, Microsoft XNA must be installed. After installing XNA, open the solution file with Visual Studio 2010 and compile the project.

If you would rather just run the compiled program I have included, simply run the executable on any Windows machine with Microsoft .Net framework version 4.0 or later.

I have also included a short sample video of me creating a simple race track and then racing on it.