15-462 Project 1: Splines

Release Date: Friday, September 18, 2009
Due Date: Tuesday, October 20, 2009

Starter Code: http://www.cs.cmu.edu/~15462/proj/p1.tar.gz

1 Overview

For this project, you will implement a few mini-projects based on the use of splines. There are 3 parts. First, you will implement the rendering of a few different kinds of 2D splines. Second, you will use a 3D spline to deform and animate a triangle mesh. Finally, you will use a spline to render a roller coaster track, running the camera along the track. You will also be introduced to OpenGL texture mapping.

It is highly advised that you start early, as this project is quite involved. Consider setting checkpoints for yourself for each part.

2 Submission Process and Handin Instructions

Failure to follow submission instructions will negatively impact your grade.

1. Your handin directory may be found at /afs/cs.cmu.edu/academic/class/15462-f09/handin/andrewid/p1/. All your files should be placed here. Please make sure you have a directory and are able to write to it well before the deadline. We are not responsible if you wait until 10 minutes before the deadline and run into trouble. Also, remember that you must run aklog cs.cmu.edu every time in order to read from/write to your submission directory.

2. You should submit all files needed to build your project, as well as any textures, models, tracks, or screenshots that you used or created. Your deliverables include:

   - `src/` folder with all `.cpp` and `.hpp` files.
   - Makefile and all `*.mk` files
   - `p1.txt`
   - Any models/textures/tracks need to run your code.
Submitting the `include` and `lib` directories are optional. Be aware that you have a limit to your AFS space, so do not submit an unreasonably large number of models or images.

3. Please **do not** include:
   - The `bin/` folder or any `.o` or `.d` files.
   - Executable files

4. Do not add levels of indirection when submitting. For example, your `makefile` should be at `.../andrewid/p1/Makefile`, *not* `.../andrewid/p1/myp1/Makefile` or `.../andrewid/p1/p1.tar.gz`. Please use the same arrangement as the handout.

5. We will enter your handin directory, add the `include` and `lib` directories, and run `make`, and it should build correctly. **The code must compile and run on the WeH 5336 cluster machines.** Be sure to check to make sure you submit all files and that it builds correctly.

### 3 Required Tasks

We break down the required tasks in more detail in each section on the projects, first covering the overall requirements. Consult sections 6, 7, and 8 for each part’s specific requirements.

In general, for your program, you must:

- Submit one or screenshots of each program’s renderings.
- Use good code style and document well. We **will** read your code.
- Fill out `p1.txt` with details on your implementations.

There is also opportunity for up to 5% extra credit by implementing things above the minimum requirements. See section 10 for details.

`p1.txt` should contain a description of your implementation, along with any information about your submission of which the graders should be aware. Provide details on which algorithms you used for the various portions of the lab. Essentially, if you think the grader needs to know about it to understand your code, you should put it in this file. You should also note which source files you edited.

Examples of things to put in `p1.txt`:

- Mention parts of the requirements that you did not implement and why.
- Describe any complicated algorithms used or algorithms that are not described in the book/handout.
- Justify any major design decisions you made, such as why you chose a particular algorithm.
- List any extra work you did on top of basic requirements of which the grader should be aware.
4 Starter Code

It is recommended that you begin by first reviewing the starter code is provided. Most is the same as the previous project, with some minor adjustments.

4.1 Building and Running the Code

The code is designed to run and build on the SCS Linux machines. Please be aware that X11-forwarding has several issues with OpenGL, and that this is not a supported method. The makefile builds 3 executables, one for each project. README.txt in the starter code contains more detailed build instructions.

We have also provided a Visual Studio 2008 solution, though it will take a bit of effort to get working since the programs have required command-line arguments. More details are in the starter code. If you Windows, your project still must build and run on the 5336 machines, so you will still have to go there to test it before submitting.

4.2 What You Need to Implement

The code that you are required to implement is located in 3 files, all of the form pi_*.cpp. There are several empty shell functions which you need to modify. The specification for each function is in the source file, and relevant types are generally in the corresponding header file. You may additionally edit any other source files in the handout, though you must keep the basic program behavior the same. To add additional source files, edit the lists in sources.mk.

5 Grading: Visual Output and Code Style

Your project will be graded both on the visual output (both screenshots and running the program) and on the code itself. We will read the code.

In this assignment, part of your grade is on the quality of the visuals, in addition to correctness of the math. So make it look nice. Extra credit may be awarded for particularly good-looking projects. See section 10 for more extra credit opportunities.

Part of your grade is dependent on your code style, both how you structure your code and how readable it is. You should think carefully about how to implement the solution in a clean and complete manner. A correct, well-organized, and well-thought-out solution is better than a correct one which is not. As before, we will be looking for correct usage of the C/C++ language.

Since we read the code, please remember that we must be able to understand what your code is doing. So you should write clearly and document well. If the grader cannot tell what you are doing, then it is difficult to provide feedback on your mistakes.
6 Part 1: An Interactive 2D Spline Renderer

The first part of the assignment involves rendering various types of 2D splines. You must implement a basic spline evaluation function to render 2D curves based on given control points. You will experiment with 3 types of splines: Hermite, Catmull-Rom, and Bezier.

6.1 Requirements

**Input:** The starter code provides a program to interactively place control points by clicking the mouse. It then renders those control points.

**Output:** You must render a curve of a specified kind of spline using the control points.

**Requirements:**

- Implement `p1_2d_render` in `p1_2d.cpp`. The specification is in the source file.
- Render Hermite, Catmull-Rom, and Bezier splines using a given list of control points.
- Use recursive subdivision to ensure the curve is smooth enough.

6.2 Using the Program

The program takes a single argument, the type of spline. Valid options are `hermite`, `catmull-rom`, and `bezier`. Initially there are zero control points. Each time the user clicks the window, a control point is added at the location of the click. Your job is to render a spline using those points. You can also change the type of spline at runtime using the right-click menu.

6.3 Implementation Details

6.3.1 Piecewise Polynomials

As described in the lecture notes, in this lab we are dealing with cubic, piecewise polynomials over a vector space (either $\mathbb{R}^2$ or $\mathbb{R}^3$). So when we say “spline,” we are specifically referring to a piecewise cubic polynomial.

For this part of the lab, you only really need to implement 2 parts of a spline:

- Conversion from control points to polynomial coefficients.
- Evaluation of the polynomial at a certain time.

6.3.2 Interpreting the Control Points

Details on converting 4 control points to polynomial coefficients for each type of spline are in the lecture notes.

The way we get 4 control points in the first place is by sliding a “window” down the given list of control points. Let’s look at Catmull-Rom as an example.
Say our given list is
\[ c_1, c_2, c_3, \ldots, c_n \]

For the first segment of the spline, we use \((c_1, c_2, c_3, c_4)\). For the next segment, we use \((c_2, c_3, c_4, c_5)\). This continues until we run out of points, so the last segment will be \((c_{n-3}, c_{n-2}, c_{n-1}, c_n)\).

Note that the amount we shift the window for each segment varies for the different kinds of splines, since each type interpolates different points. For Catmull-Rom the shift width is 1. For Bezier, the shift width is 3. Hermite is a bit different, since we need both positions and tangents. So every even-indexed control point \(c_{2k}\) is a position, and its corresponding tangent is \(c_{2k+1} - c_{2k}\). Therefore, the shift width for Hermite is 2.

**Note:** Catmull-Rom also requires a tension parameter \(\tau\). You may use whatever you like for \(\tau\). A suggestion is .5.

### 6.3.3 Rendering via Subdivision

Details on spline rendering are in the lecture notes. We require you to use recursive subdivision to guarantee that the curve is smooth enough. “Smooth enough” vaguely means that each rendered segment is no more than 1 or 2 pixels in length. Note that we scale the camera for you such that one OpenGL “unit” is one pixel.

![Figure 1: Example part 1 output: Catmull-Rom splines](image-url)
7 Part 2: Model Deformation and Animation Using 3D Splines

The second part of the assignment involves animating a triangle mesh using 3D splines. You will implement a simple application of 3D splines to do a basic animation, in this case animating a snake. For this part of the program, we provide you with a model and a list of control points, and you must deform the model using those control points.

The premise is simple; you must represent each vertex of the mesh in terms with which you can use the spline to compute a new position. So a vertex will be extruded from the spline somehow, and should follow the spline around as the spline moves.

7.1 Requirements

Input: We provide you with a model and an animated set of control points.
Output: You must deform and render this model using the control points, with texturing and basic lighting and shading.

Requirements:

- Implement the `p1_model_*` functions in `p1_model.cpp`. The specification is in the source file.
- Use Hermite splines to deform the provided model using the given control points and up vector for each provided animation.
- Texture and light the model with the provided texture.

7.2 Using the Program

The program takes 2 required arguments and a 3rd optional argument. The first option specifies a `.obj` file to use as the model. The second is a `.png` to use as the texture. The third argument is the name of the animation you want.

The model and texture we have provided you are `models/snake.obj` and `models/snake.png`. The animations we provide you are listed in `p1_model.cpp`. There are 3 of them: `static`, `simple`, and `default`. You may add additional animations if you like.

The user can move the camera with the mouse, shift, and control. The program runs an update/render loop, calling your update and render functions every frame. The update function should tick the animation and create new control points, and the render function should deform the model and render it.

7.3 Implementation Details

7.3.1 Transforming the Model

We provide you with a model, similar to project 0. This time, we load the normals for you, so you need to transform the vertex positions and normals using the spline.
The control points are for Hermite splines. You should segment them the same way as you did in part 1. Each control point has a position, derivative, and up vector. Your model should meet the given up vector at each control point, and the up vector should transition smoothly between points.

You may use any algorithm that you wish, as long as it meets the requirements. The model in the file is placed as if the spline ran along the $Z$ axis from 0.0 to 4.0. Details of model transformation are given in section 9.

**Note:** You may find it very useful for debugging to render the spline, control points, derivatives, and up vectors using `GL_POINTS` and `GL_LINES`. However, these should not be rendered in your final submission.

### 7.3.2 Texturing and Lighting

The model also has a texture that you must render. Details on 2D texture mapping are given in the Red Book. We provide you with an image-loading function, `imageio_load_image`, which loads a PNG file into an array of bytes in RGBA format. The function is in the header `application/imageio.hpp`.

The texture’s filename is specified as a member of the `Model` class, so you can access it at runtime from there.

Texture coordinates are given for the model along with the positions and normals. Therefore, you will also need to start sending the texture coordinate attribute data along with the vertex and normal data. Consult the Red Book for details.

You must also again provide some basic lighting and shading on the model. Since we have a texture, you can leave the diffuse, specular, and ambient materials of the model at white, but you still must create a light for the scene.

![Figure 2: Example part 2 output](image-url)
8 Part 3: Roller Coaster: Model Generation and Camera Animation Using 3D Splines

The last part of the assignment involves generating a model along 3D spline, and using the splines to move the camera along the spline as if it were a track. We provide you with a list of control points, and you must generate a looping track along those points, then move the camera along the track.

8.1 Requirements

Input: The starter code loads a list of Catmull-Rom control points from a file.
Output: You must render a coaster track using those points, and then have the camera move along the track in a physically realistic manner.

Requirements:

- Implement the p1_coaster_* functions in p1_coaster.cpp. The specification is in the source file.
- Use Catmull-Rom splines to generate a coaster track using the given control points and up vectors.
- Light the scene and texture or color the track in some way.
- The up vectors need to be $C^0$ continuous and should not twist excessively.
- Make the camera follow the coaster track, pointing along the derivative, using physically realistic motion.
- Render some kind of background for the scene.

8.2 Using the Program

The program takes 1 argument, a text file containing a list of control points, which is parsed and given to the initialization function. Each frame, the update and render functions are called, which should move the camera along the track and render the scene from the tracks perspective.

Tracks are looped. The track should form a closed loop when rendered.

We provide several sample tracks, which are in the tracks/ folder. Your program must work with all of the given tracks. You are also encouraged to create additional tracks of your own.

8.3 Implementation Details

8.3.1 Splines

The control points are for Catmull-Rom splines (different than part 2!). You should segment them the same way as you did in part 1. Since the track is looped, you must have the control points wrap around into a circular list so that the coaster fits flush together. The tension parameter $\tau$ is also given to you.
Each control point contains a position and an up vector. As before, you must hit the up vector at each control point, and the up vectors must change smoothly between points. That is, your coaster should not have sudden snaps and changes in the up vector, and should not twist horribly in on itself. You may use any method that you come up with as long as it is continuous and doesn’t have excessive unnecessary twisting. You must achieve at least $C^0$ continuity. Section 9.3 provides some possible methods, though you may use your own.

8.3.2 Generating a Track

The math behind the geometry deformation is very similar to part 2. See section 9 for more information. The information on transforming vertices is the same. However, instead of having to compute the spline representation from a given model, you make it procedurally.

Another difference is that we want the track to be evenly spaced in world, which is not the same as evenly spaced time steps along the spline. So you will probably not be evaluating the spline at even time steps, but rather computing the time step so that world spacing is even.

We make no requirements on what the track should look like, other than it should be some kind of repeating geometry that plausibly could be a track. You must color it in some way, either using any textures you find or create, or just with colors. Be creative and make something interesting.

8.3.3 Moving the Camera

Setting the camera at a specific point on the spline is simple. Use the spline’s derivative as the camera’s direction and the spline’s up vector as the camera’s up vector. You should offset the position slightly so that the camera is slightly above the track rather than inside of it. You may also want to tilt the direction down slightly to look at the track rather than straight ahead. The camera should run continuously around the loop.

The camera’s speed should be done in a physically realistic manner, ignoring friction. Just apply conservation of energy. So pick some height where the kinetic energy is 0 that is above the maximum height of the track. Then compute the kinetic energy at the current height to get the current camera speed. This baseline height can be whatever you like, as long as it doesn’t cause the camera to move so fast that you can’t clearly see the track. Therefore, it should be relative to the maximum track height.

8.3.4 Rendering a Background

The last requirement is to render some kind of background for your coaster. We don’t care about the specifics, as long as it is obvious where the world up vector is as the camera moves around.

One suggested background is a ground plane with some kind of sky, perhaps a skydome or a skybox. Be creative and choose whatever background you find interesting.
9 Transforming Vertices Using 3D Splines

Here we present some supplemental explanation of the mathematics behind 3D splines, coordinate frames, and deforming vertices using a spline. You may find this information helpful in your implementation, though it is not required reading.

9.1 3D Splines and Coordinate Frames

9.1.1 Spline as Position over Time

Recall that a spline is simply a parametric piecewise polynomial function. For model deformation, however, the details of the type of spline are unimportant. In fact, we don’t even care that it is a polynomial; all that we need to know is that it is a continuously differentiable function from some range in the reals to 3D vectors. That is, we consider a spline $S$ as

$$S : [a, b] \rightarrow \mathbb{R}^3, \quad S \in C^1$$

for some fixed $a, b$. By convention, we will usually refer to the domain by $t$, for time. This does not stand for physical time, but merely the distance along the spline’s path.

With that definition, we have enough to render the spline as a curve, as you did in part 1. However, we want to extrude points from the surface of the spline, so this information is not enough. We need to know which direction is “up” at any given point on the spline, so that we know in which direction the geometry should stick out from the spline body. If we aren’t careful, we could end up with some nasty twisting of our model.

9.1.2 Coordinate Frames

So we also need to have a sense of the coordinate frame at any point on the spline. Which way is forward, and which way is up? That is, we want an orthonormal basis of $\mathbb{R}^3$ to represent the major axes, rotated into the frame of the spline. We can just use the derivative $S'$ as the forward direction, but we’ll need a new piece of information for the up vector.

So let us redefine our notion of a 3D spline to include both a position and an up vector, where the position and up are both continuously differentiable. That is, the spline $F$ is

$$F : [a, b] \rightarrow \mathbb{R}^3 \times \mathbb{R}^3, \quad s \in C^1.$$  

The first term is a position, and the second term is the up vector at that position. Note that using this, we can define an orthonormal basis $(U_t, V_t, W_t)$ to use as our coordinate frame. We let $V_t$ be the up vector at $t$, $U_t$ be the derivative of the position (tangent vector) at $t$, and then $W_t$ is the cross product $U_t \times V_t$.

Note: This assumes that the up vector is already orthogonal to the tangent.
But you can still generate an orthonormal basis rather easily even if this is not true, which you may find useful to do in your code.

Note that both position and up should be continuously differentiable to prevent sudden twisting in the coordinate frames. See section 9.3 for information about making the up vectors continuous.

Let us adopt a few naming conventions. Given any spline, let

- $S(t)$ be the position of the spline at time $t$.
- $S'(t)$ be the tangent or derivative of position of the spline at time $t$.
- $V(t)$ be the up vector of the spline at time $t$.
- $(U_t, V_t, W_t)$ be the orthonormal basis that forms a coordinate frame for the spline, computed as described above.

### 9.2 Vertex Transformations

#### 9.2.1 Representing Vertices and Forward Transformation

Now armed with the notion of coordinate frames along a spline, we are ready to consider how to deform a model. Traditionally, we represent a vertex using its position $P = (x, y, z)$, $P \in \mathbb{R}^3$. When deforming a vertex along a spline, how could we define a vertex? We will present one possible method that we suggest you use in your implementation.

Since we want the vertex to “follow” the spline, one natural parameter is the distance along the spline, $t$. For example, a vertex right in the middle of the spline would have $t = (b - a)/2$. What about the others? Well, we need to know in which direction and how far along $W_t$ and $V_t$ the vertex lies. So let our new vertex representation be $P' = (t, (v, w))$, $P' \in [a, b] \times \mathbb{R}^2$.

Then the function $A$ that takes $P'$ to $P$ is simply

$$A(t, (v, w)) = S(t) + V_t v + W_t w$$  \hspace{1cm} (1)$$

That is, we extrude the vertex by $(v, w)$ from the spline’s curve at $t$ using the coordinate frame at $t$.

#### 9.2.2 Backward Transformation

What about the other direction, $P$ to $P'$? The backward transformation is a bit more difficult, at least without assuming something more about the spline. Fortunately, the model is defined on the simplest spline of all (a straight line), which makes the job much simpler.

We wish to compute $t$ and $(v, w)$ given $P = (x, y, z)$. Since we are doing the backward transformation from a straight line along an axis, $t$ is simply the distance of the vertex along that axis and fit into the range $[a, b]$.

Now let $G = P - S(t)$. This is the vector that points from the spline’s curve to $P$. Furthermore, it is orthogonal to $S'(t)$ and therefore lies in the plane with $V_t$ and $W_t$, so we can represent it in terms of coefficients of $V_t$ and $W_t$, which is our $(v, w)$. 

11
In your case, the model runs along the $Z$ axis from $a = 0.0$ to $b = 4.0$, with the up being the $Y$ axis. That makes this transformation relatively simple to implement, since for $p = (x, y, z)$, you have

\[ t = z - a \]
\[ w = x \]
\[ v = y \]

### 9.2.3 Normal Transformation

The last thing we must encode are the normal transformations. We want the normal to have the same angle relative to the closest point on the spline body at all times. Therefore, we can store the normal as a rotation along a fixed vector, specifically the $G$ defined above. Let its normal be $N$. Then we can store the normal $M$ as a rotation from $G$ to $N$. That is, the matrix such that

\[ N = MG \]

which can be easily computed. We know the axis of rotation is $G \times N$, the cosine of the angle of rotation is $G \cdot N$ and the sine of the angle of rotation is $|G \times N|$ (assuming both are unit vectors).

**Note:** We provide you with a quaternion class that can be used in place of a rotation matrix. You may find this useful.

Once we have $M$, to get the normal, all you have to do is compute a $G'$ to rotate, where $G'$ similarly defined, only at the computed $P'$. we can use $G' = Vt v + Wt w$, and thus:

\[ N' = M(Vt v + Wt w) \]

### 9.3 Choosing the Up Vector

Making the coordinate frame along the spline continuous is important so that the model doesn’t have a sudden change in orientation. Further more, we want the changes to be as smooth as possible to prevent twisting.

We give you the up vector at each control point. You need to therefore find a way to compute the up vector between two control points such that it is continuous. $C^1$ continuity is preferable, though $C^0$ continuity is still better than none.

One simple method is just to use some kind of interpolation of the up vectors. Of course, this doesn’t guarantee that the results will be orthogonal to the derivative, so you’ll have to compute the closest orthogonal vector. However, this can cause massive twisting in some pathological cases.
9.3.1 Rotation Minimizing Frames

It must be more than just continuous. We also want to keep the amount of twisting to a minimum. Of course, you’ll have some twisting since you have to meet the up vectors at the control points. But we’d like to avoid unnecessary twists.

For each spline and initial up vector, we can compute a rotation minimizing frame, which is the function from time to up vector such that the twisting along the spline is minimal. There are several very fast ways to approximate this discretely. This paper, www.cs.hku.hk/research/techreps/document/TR-2007-07.pdf, demonstrates one such method.

It is possible to use the RMF to compute intermediate up vectors for your project that are both continuous and keep twisting minimal. We will leave the details for you to figure out.

10 Extra Credit

This assignment leaves quite a bit of room for creative freedom and with five weeks, it’s possible to go quite far with your implementation of the assignment. For any additional work that you may do on the project, it is possible to get up to five percent extra credit for second and third parts of the project. We are not offering any extra credit for the first part.

Extra credit will be awarded at the discretion of the TAs. As always, please make sure to document any additional work that you may do in your p1.txt. Some ideas and suggestions for extra credit are:

10.1 Part 2

• Create a new model and texture besides just a snake.
• Create an interesting animation for the snake. You could animate the snake to perform backflips, for instance. The more complicated, the better.

10.2 Part 3

• Find a way to ensure you up vectors are $C^1$ continuous.
• Add additional models to your scene. For example, you could create cars that could run along the track.
• Design extra tracks and/or make the coaster work with different types of splines.
• Add a complex and interesting backdrop for you coaster. Your own imagination is the limit.
• Use modern GPU features to create interesting special effects. For example, motion blur or fire.
11 Words of Advice

• Start early. The mathematics involved can be confusing, and there is quite a bit of coding involved. You are given over a month for this project; do not expect to be able to complete it in a day or two.

• Bear in mind that you have a homework and a midterm during the latter half of the project’s span, and that you probably don’t want to be doing both of those and this entire project at once—even more reason to start early.

• Begin with part 1, as it provides a good foundation for the next 2 parts.

• We suggest you set checkpoints for yourself, as this is essentially 3 separate (albeit closely related) projects. Our suggestion is to allocate 1 week to part 1, 2 weeks to part 2, and 2 weeks to part 3.

• Read the entire handout carefully, and make sure you understand the mathematics involved before beginning an implementation.

• These parts require a lot of the same functionality, which means you can have a lot of code reuse. We highly suggest that you carefully consider how to organize the code to reduce code repetition. Remember that part of your grade is dependent on code organization.

• Make sure you have a submission directory that you can write to as soon as possible. Notify course staff if this is not the case.

• Any Linux computer on campus may be used to build the project. However, you must still make sure that it compiles on the 5336 cluster machines.

• If you use Windows to implement the project, be sure to test on the Linux machines. The compilers are not quite the same, and certain things that compile with MSVC do not compile or behave differently with GCC.