#15: Final Project: Roller Coaster!

CSE167: Computer Graphics
Instructor: Ronen Barzel
UCSD, Winter 2006
Roller coasters...
Final project

- Build a roller coaster, with a car riding on it
- More open-ended than previous projects
- Be creative!
  - We do have some specific capabilities/features we’ll look for
  - But we’ll be impressed by “style” too!
Final project

- More on your own than previous projects
- No “base code”
  - though you may use base code from previous assignments

- Today I’ll go over the key techniques
Roller coaster steps

(see web page for details)

1. Make a piecewise-cubic curve
2. Create coordinate frames along a piecewise-cubic curve
   - will be used to orient along the path
   - include a roll angle at each curve point, to adjust tilt
3. Create a swept surface
   - the actual track along the of the roller coaster
4. Design and build the roller coaster track
5. Run a car on the track
Step 1. Piecewise-Cubic Curve

- Specify as a series of points
- Will be used for the path of the roller coaster
Step 2. Coordinate Frames on curve

- Describes orientation along the path
Step 2b. Tweak coordinate frames

- Control lean of coordinate frames
- Specify “roll” angle offset at each control point
Step 3a. Sweep a cross section

- Define a cross section curve (piecewise-linear is OK)
- Sweep it along the path, using the frames
3b. Tessellate

- Construct triangles using the swept points
- (sweep more finely than this drawing, so it’ll be smoother)
4. Run a car on the track
Step 1. Piecewise-Cubic Curve

- Specify as a series of points
- Will be used for the path of the roller coaster
Piecewise-Cubic Curve

- Requirements:
  - Given an array of N points
  - Be able to evaluate curve for any value of t (in 0…1)
  - Curve should be C1-continuous

- Best approach: define a class, something like

```java
class Curve {
    Curve(int Npoints, Point3 points[]);
    Point3 Eval(float t);
}
```
Piecwise-Cubic Curves

Three recommended best choices:

- Bézier segments
  - Advantages: You did a Bézier segment in Project 5
  - Disadvantages: Some work to get C1 continuity
    - Will discuss technique to do this

- B-Spline
  - Advantages: simple, always C1 continuous
  - Disadvantages: New thing to implement, doesn’t interpolate

- Catmull-Rom spline
  - Advantages: interpolates points
  - Disadvantages: Less predictable than B-Spline. New thing to implement
Piecewise Bézier curve review

- We have a Bézier segment function $Bez(t,p_0,p_1,p_2,p_3)$
  - Given four points
  - Evaluates a Bézier segment for $t$ in $0…1$
  - Implemented in Project 5

- Now define a piecewise Bézier curve $x(u)$
  - Given $3N+1$ points $p_0,…,p_{3N}$
  - Evaluates entire curve for $u$ in $0…1$
  - Note: In class 8, defined curve for $u$ in $0…N$
    - Today will define it for $u$ in $0…1$
Piecewise Bézier curve: segments

- Given $3N + 1$ points $p_0, p_1, \ldots, p_{3N}$
- Define $N$ Bézier segments:
  
  $x_0(t) = \text{Bez}(t, p_0, p_1, p_2, p_3)$
  
  $x_1(t) = \text{Bez}(t, p_3, p_4, p_5, p_6)$
  
  $\vdots$
  
  $x_{N-1}(t) = \text{Bez}(t, p_{3N-3}, p_{3N-2}, p_{3N-1}, p_{3N})$
Piecewise Bézier curve

\[ x(u) = \begin{cases} 
  x_0(Nu), & 0 \leq u \leq \frac{1}{N} \\
  x_1(Nu - 1), & \frac{1}{N} \leq u \leq \frac{2}{N} \\
  \vdots & \vdots \\
  x_{N-1}(Nu - (N - 1)), & \frac{N-1}{N} \leq u \leq 1 
\end{cases} \]

\[ x(u) = x_i(Nu - i), \text{ where } i = \lfloor Nu \rfloor, u < 1; \]

\[ = \text{Bez}(Nu - i, p_{3i}, p_{3i+1}, p_{3i+2}, p_{3i+3}) \]

\[ x(1) = p_{3N} \]

\[ = \text{Bez}(1, p_{3i}, p_{3i+1}, p_{3i+2}, p_{3i+3}) \]
Piecewise Bézier curve

- 3N+1 points define N Bézier segments
- \( x(i/N) = p_{3i} \)
- \( C^0 \) continuous by construction
- \( G^1 \) continuous at \( p_{3i} \) when \( p_{3i-1}, p_{3i}, p_{3i+1} \) are collinear
- \( C^1 \) continuous at \( p_{3i} \) when \( p_{3i} - p_{3i-1} = p_{3i+1} - p_{3i} \)

\[ \begin{align*}
  p_0 & \quad p_1 \\
  p_2 & \quad p_3 \\
  p_4 & \quad p_5 \\
  p_6 &
\end{align*} \]

\( C^1 \) discontinuous

\[ \begin{align*}
  p_0 & \quad p_1 \\
  p_2 & \quad p_3 \\
  p_4 & \quad p_5 \\
  p_6 &
\end{align*} \]

\( C^1 \) continuous
Tangent to piecewise Bézier curve

- Tangent to the piecewise curve
  - same as the tangent to each segment (from Project 5)
    - mathematically speaking, needs to be scaled
    - in practice we often normalize anyway
  - At the shared points, the tangents on the two sides might not agree
    - Unless we arrange for C1 continuity

\[
\mathbf{x}(u) = \text{Bez}(t, \mathbf{p}_{3i}, \mathbf{p}_{3i+1}, \mathbf{p}_{3i+2}, \mathbf{p}_{3i+3})
\]
\[
\mathbf{x}'(u) = N \ \text{Bez}'(t, \mathbf{p}_{3i}, \mathbf{p}_{3i+1}, \mathbf{p}_{3i+2}, \mathbf{p}_{3i+3})
\]

where \[
\begin{aligned}
&u < 1: \ i = \lfloor Nu \rfloor \text{ and } t = Nu - i \\
&u = 1: \ i = N - 1 \text{ and } t = 1
\end{aligned}
\]
Piecewise Bézier curves

- Inconveniences:
  - Must have 4 or 7 or 10 or 13 or … (1 plus a multiple of 3) points
  - Not all points are the same
  - Have to fiddle to keep curve C1-continuous
Making a C1 Bézier curve

- A hack to construct a C1-continuous Bézier curve
  - Actually, this is essentially implementing Catmull-Rom splines
  - Instead, could just implement Catmull-Rom splines directly
    • Same amount of work as implementing B-Spline

- Given M points:
  - want a piecewise-Bézier curve that interpolates them
  - Bézier segments interpolate their endpoints
  - Need to construct the intermediate points for the segments
Auto-creation of intermediate points

- Given $M$ original points
  - Will have $M-1$ segments
  - Will have $N=3(M-1)+1$ final points
Auto-creation of intermediate points

- Need tangent directions at each original point
  - Use direction from previous point to next point
Auto-creation of intermediate points

- Introduce intermediate points
  - Move off each original point in the direction of the tangent
  - “Handle” length = $1/3$ distance between previous and next points
  - special-case the end segments.
    - or for closed loop curve, wrap around
Auto-creation of intermediate points

- Resulting curve:
Auto-creation of intermediate points

- Summary of algorithm:
  Given $M$ originalPoints
  Allocate newPoints array, $N = 3*(M-1)+1$ new points
  for each original point $i$
    $$p = \text{originalPoints}[i]$$
    $$v = \text{originalPoints}[i+1]-\text{originalPoints}[i-1]$$
    $$\text{newPoints}[3*i-1] = p-v/6$$
    $$\text{newPoints}[3*i] = p$$
    $$\text{newPoints}[3*i-1] = p+v/6$$

- Of course, must be careful about start and end of loop
- Can allow the curve to be a closed loop: wrap around the ends
Other option... B-spline

- Need at least 4 points, i.e. N+3 points
- B-spline curve uses given N+3 points as is
  - Always C2 continuous
    - no intermediate points needed
  - Approximates rather than interpolates
    - doesn’t reach the endpoints
    - not a problem for closed loops: wrap around.
- simple algorithm to compute
  - (may need to extend matrix library to support Vector4)
Blending Functions Review

- Evaluate Bézier using “Sliding window”
  - Window moves forward by 3 units when $u$ passes to next Bézier segment
  - With $3N+1$ points, $N$ positions for window, ie. $N$ segments
  - Evaluate matrix in window:

\[
x(u) = \begin{bmatrix} p_{3i} & p_{3i+1} & p_{3i+2} & p_{3i+3} \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix}
\]

where

\[
\begin{align*}
& u < 1: \quad i = \lfloor Nu \rfloor \text{ and } t = Nu - i \\
& u = 1: \quad i = N - 1 \text{ and } t = 1
\end{align*}
\]
B-spline blending functions

- Still a sliding “window”
- Shift “window” by 1, not by 3
- With N+3 points, N positions for window, i.e. N segments

\[
x(u) = [p_i \ p_{i+1} \ p_{i+2} \ p_{i+3}] \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 0 & 4 \\ -3 & 3 & 3 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix}
\]

where

\[
\begin{align*}
&u < 1: \ i = \lfloor Nu \rfloor \text{ and } t = Nu - i \\
&u = 1: \ i = N - 1 \text{ and } t = 1
\end{align*}
\]
Tangent to B-spline

- Same formulation, but use $T'$ vector (and scale by $N$)

\[ \mathbf{x}(u) = \left[ \mathbf{p}_i \mathbf{p}_{i+1} \mathbf{p}_{i+2} \mathbf{p}_{i+3} \right] \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 0 & 4 \\ -3 & 3 & 3 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix} \]

\[ \mathbf{x}'(u) = \left[ \mathbf{p}_i \mathbf{p}_{i+1} \mathbf{p}_{i+2} \mathbf{p}_{i+3} \right] \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 0 & 4 \\ -3 & 3 & 3 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 3t^2 \\ 2t \\ 1 \\ 0 \end{bmatrix} \]

where
\[ u < 1: \quad i = \lfloor Nu \rfloor \quad \text{and} \quad t = Nu - i \]
\[ u = 1: \quad i = N - 1 \quad \text{and} \quad t = 1 \]
Catmull-Rom

- Use same formulation as B-spline
  - But a different basis matrix
- Interpolates all its control points
  - Except it doesn’t reach the endpoints
  - Add extra points on the ends, or wrap around for closed loop

\[ x(u) = \frac{1}{2} \begin{bmatrix} -1 & 2 & -1 & 0 \\ 3 & -5 & 0 & 2 \\ -3 & 4 & 1 & 0 \\ 1 & -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix} \]

where \( u < 1: \quad i = \left\lfloor Nu \right\rfloor \) and \( t = Nu - i \)

\[ u = 1: \quad i = N - 1 \) and \( t = 1 \)

\[ x'(u) = \frac{1}{2} \begin{bmatrix} -1 & 2 & -1 & 0 \\ 3 & -5 & 0 & 2 \\ -3 & 4 & 1 & 0 \\ 1 & -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 3t^2 \\ 2t \\ 1 \\ 0 \end{bmatrix} \]

where \( u < 1: \quad i = \left\lfloor Nu \right\rfloor \) and \( t = Nu - i \)

\( u = 1: \quad i = N - 1 \) and \( t = 1 \)
Step 1, Summary

- We can define a curve:
  - Given N points
  - We know how to evaluate the curve for any u in 0..1
  - We know how to evaluate the tangent for any u in 0..1
  - Might be implemented via Bézier, B-Spline, Catrom

- Bundle in a class

```java
class Curve {
    Curve(int Npoints, Point3 points[]);
    Point3 Eval(float u);
    Vector3 Tangent(float u);
}
```
Step 2. Coordinate Frames on curve

- Describes orientation along the path
Finding coordinate frames

- To be able to build the track, we need to define coordinate frames along the path.
  - We know the tangent, this gives us one axis, the “forward” direction
  - We need to come up with vectors for the “right” and “up” directions.
- There’s no automatic solution
- Most straightforward:
  - Use the same approach as camera lookat
  - Pick an “up” vector.
Constructing a frame

- Same as constructing an object-to-world transform

Given that we can compute position \( x(t) \) and tangent \( \ddot{x}(t) \)
Given a global "up" vector \( \vec{u} \)
Assume that we want
  "forward" direction to be the y axis,
  "right" to be the x axis
  "up" to be the z axis

To construct the frame matrix \( F(t) \), fill the \( \vec{a}, \vec{b}, \vec{c}, \vec{d} \) columns

\[
\begin{align*}
\vec{d} &= x(t) & \text{-- origin of the frame is at } t \text{ along the curve} \\
\vec{b} &= \frac{\ddot{x}(t)}{|\ddot{x}(t)|} & \text{-- y axis of frame is tangent, normalized} \\
\vec{a} &= \frac{\vec{b} \times \vec{u}}{|\vec{b} \times \vec{u}|} & \text{-- x axis of frame is perpendicular to forward and global up} \\
\vec{c} &= \vec{a} \times \vec{b} & \text{-- z axis of frame is perpendicular to forward and right}
\end{align*}
\]
Specifying a roll for the frame

- The “lookat” method tries to keep the frame upright
- But what if we want to lean, e.g. for a banked corner?
- Introduce a roll parameter
  - Gives the angle to roll about the “forward” axis.
- Want to be able to vary the roll along the path
  - Specify a roll value per control point
Computing the roll on the path

- How to compute the roll at any value of $t$?
  - just use the roll values as (1D) spline points!

Given points: $p_0, p_1, \ldots, p_{N-1}$
and rolls: $r_0, r_1, \ldots, r_{N-1}$

Compute position, tangent:

$$x(u) = Bez(Nu - i, p_{3i}, p_{3i+1}, p_{3i+2}, p_{3i+3})$$
or
$$x(u) = G_iBT_{Nu-i}$$

Compute roll:

$$r(u) = Bez(Nu - i, r_{3i}, r_{3i+1}, r_{3i+2}, r_{3i+3})$$
or
$$r(u) = [r_i \ r_{i+1} \ r_{i+2} \ r_{i+3}]BT_{Nu-i}$$

where:

$$i = \left\lfloor Nu \right\rfloor$$
Rolling the frame

To compute the final object-to-world matrix for the frame, compose the upright frame with a rotation matrix:

\[ W(t) = F(t) R_y(r(t)) \]
Curve class with frames

The class can include:

class Curve {
    Curve(int N, Point3 points[], float roll[]);
    void SetUp(Vector3 up);
    Point3 Eval(float t);
    Vector3 Tangent(float t);
    Matrix Frame(float t) { return UpFrame(t)*Roll(t); } }

private:
    Matrix UpFrame(float t);
    Matrix Roll(float t);
}
Step 3a. Sweep a cross section

- Define a cross section curve (piecewise-linear is OK)
- Sweep it along the path, using the frames
3b. Tessellate

- Construct triangles using the swept points
- (sweep more finely than this drawing, so it’ll be smoother)
The cross section

- Say we have a cross section curve with N points
  - AKA the *profile curve*
  - For simplicity, assume it’s piecewise linear
  - e.g., N=4

\[ q_0 \quad q_1 \quad q_2 \quad q_3 \]
Sweep the profile curve

- choose M how many rows to have
  - make this a parameter or value you can change easily
  - too few: not smooth enough
  - too many: too slow
- place M copies of the cross section along the path
  - e.g. M=8 (for a full track you might have M=100’s or more)
Sweep the profile curve

- We can get a coordinate frame for any $t$ along the curve
  - i.e. we have an object-to-world matrix $W(t)$ for $t$ in $0...1$

- Distribute the profile curve copies uniformly in $t$
  - For each row, transform the profile curve using the world matrix:

$$
\text{For each row } i \text{ in } 0 \text{ to } M - 1
$$
$$
\text{For each profile point } q_j \text{ for } j \text{ in } 0 \text{ to } N - 1
$$

$$
t = \frac{i}{M - 1}
$$

$$
p_{ij} = W(t)q_j
$$
Topology

- Simple rectangular topology, M*N points:

\[ P[iN + j] = W(i/(M - 1)) \cdot q_j \]
Putting a car on the track

- Again, we can use the path’s frame matrix $W(t)$
  - Probably with a local offset $M$ if needed:
    - Local-to-world = $W(t)M$

- $t$ goes from 0 to 1 and keeps cycling

- Wheels rotate as function of $t$
  - Will probably slide. Try to tweak so they look reasonable.
  - More complex to really have wheels roll on the track
Designing a roller coaster
Designing a roller coaster track

Suggestions…

- plan it out on (graph) paper first!
- top view & “unwound” elevation view
- number and label the control points
  - use a scheme that will let you go back and add more points later!
  - control point coordinates can be hard-wired in C++ source
  - when typing them in, comment them with your labels
Designing a roller coaster

- The path can be fairly simple.
  - start with something simple
  - if you have time at the end, go wild!
- The track could be...
  - a trench, like a log flume that the car slides in
  - a monorail that the car envelops
  - a pair of rails (two cross sections) for a railroad-style car
  - whatever else you come up with
- The car could be...
  - a cube with simple wheels
  - a penguin
  - A sports car that you download off the web and figure out how to read
  - whatever else you come up with
- Trestles/Supports not needed... but could be cool
- Ticket booth & Flags & Fence etc. not needed... but could be cool
- Have fun! Be creative!
Advanced Roller Coaster Topics

- (Not required, but handy for some bells & whistles)

- Arc-Length Parametrization
  - Better control over speed of car

- Parallel transport of frames
  - Make it easier to do loops
Arc-length Parametrization

- In our current formulation:
  - Position along the curve is based on arbitrary $t$ parameter
  - We animate by increasing $t$ uniformly:
    - Speed of the car isn’t constant
    - Speed of the car isn’t easily controllable
    - Car spends the same amount of time in each segment:
      - in areas where the control points are far apart, the car goes fast
      - in areas where the control points are close together, the car goes slow
Arc-length parametrization

- Specify position along the curve based on distance $s$
  - $s=0$: at start of curve
  - $s=1$: 1 unit (foot or meter or whatever) along the track
  - $s=2$: 2 units along the track
- Animate by increasing $s$ uniformly: constant speed
- Adjust speed by adjusting change in $s$ per frame
- Unaffected by where control points happen to be
Arc-length parametrization

- *Arc length* just means length along the curve
- Really, arc length re-parametrization
  - We have a parameter $t$
  - We want to express $t$ in terms of length $s$
- No simple closed form expression
- A bit of math in principle
- Then we’ll do a simple approximation of it
**Arc length**

- Define $S(t) = \text{length along the curve } x(t) \text{ from the start until } t$

\[
ds = \sqrt{dx^2 + dy^2 + dz^2} \quad \text{-- differential distance}
\]

\[
S(t) = \int_0^t \left( \frac{ds}{dt} \right) dt
\]

\[
= \int_0^t \sqrt{\left( \frac{dx}{dt} \right)^2 + \left( \frac{dy}{dt} \right)^2 + \left( \frac{dz}{dt} \right)^2} \ dt
\]

\[
= \int_0^t |\dddot{x}(t)| \ dt
\]
Reparameterize

We have:
\[ x(t) = \text{our curve parametrized by } t \]
\[ S(t) = \text{arc length along } x(t) \text{ up to } t \]

We want:
\[ x_{arc}(s) = \text{our curve re-parametrized by } s \]

\[ x_{arc}(s) = x(t(s)), \text{ where } t(s) = S^{-1}(s) \]

We need:
the inverse length function, \( S^{-1}(s) \)
Arc-length reparameterization

- Exact computation of length & inverse is too hard

- Simple approximation for arc length: *chord length*
  - Sample $M$ times, uniformly (sufficiently large, e.g. $M=10N$)
  - Accumulate incremental length into array of length values:
    \[
    S_i = S_{i-1} + \left| \mathbf{x}\left( \frac{i}{M} \right) - \mathbf{x}\left( \frac{i-1}{M} \right) \right|
    \]

- Simple approximation to invert:
  Given $s$, where $0 \leq s \leq S_M$
  Find neighboring samples: $i$ such that $S_i < s < S_{i+1}$
  Use linear interpolation between the samples:
  \[
  t = \frac{1}{M} \left( i + \frac{s - S_i}{S_{i+1} - S_i} \right)
  \]
Parallel transport of frames

- Using the “lookat” method with a global “up” vector:
  - Trouble if the “forward” direction lines up with the “up” direction
    - No solution when it lines up
    - Solution tends to spin wildly when forward is very close to up
    - e.g. if track goes straight up or straight down
      - such as would happen going into or out of a loop

- Possible solution:
  - Specify an “up” vector per point
    - Replaces of the roll vector
    - Can compute spline on per-point up vectors
  - This would work, but is cumbersome for the user

- Alternative:
  - Start with a good frame, carry it with you along the path
Parallel Transport

- User specifies initial “up” vector for $t=0$ (or $s=0$)
  - Construct an initial frame using the lookat method

- Compute subsequent “up” vectors incrementally
  - Directly for each sample when sampling curve for tessellation; or
  - In advance when defining the curve:
    - Compute and store “up” vector at each control point
    - Use the stored “up” vectors as control points for an “up” vector spline

- Given the tangent and an “up” vector at each point
  - Can construct a frame using the lookat method
  - None of our stored “up” vectors will be parallel to the tangent.
Parallel Transport

- Given at \( t = t_i \):
  \[
  \text{tangent } \vec{v}_i = \frac{\vec{x}'(t_i)}{\| \vec{x}'(t_i) \|},
  \]
  
  "up" vector \( \vec{u}_i \),

- For next sample \( t = t_{i+1} \):
  - Compute \( \vec{v}_{i+1} = \frac{\vec{x}'(t_{i+1})}{\| \vec{x}'(t_{i+1}) \|} \)
  - Compute rotation that takes \( \vec{v}_i \) to \( \vec{v}_{i+1} \):
    \[
    \text{axis } \vec{a} = \frac{\vec{v}_i \times \vec{v}_{i+1}}{\| \vec{v}_i \times \vec{v}_{i+1} \|}
    \]
    
    angle \( \theta = \tan^{-1} \left( \frac{\vec{v}_i \times \vec{v}_{i+1}}{\vec{v}_i \cdot \vec{v}_{i+1}} \right) \)
  - Use that rotation to take \( \vec{u}_i \) to \( \vec{u}_{i+1} \):
    \[
    \vec{u}_{i+1} = R(\vec{a}, \theta) \vec{u}_i
    \]

- Works if curve doesn't turn 180° between samples
- Can also include (incremental) roll value at each sample:
  \[
  \vec{u}_{i+1} = R(\vec{v}_{i+1}, r_{i+1}) R(\vec{a}, \theta) \vec{u}_i
  \]
Done

- Enjoy your projects!
- Next class: guest lecture on game design