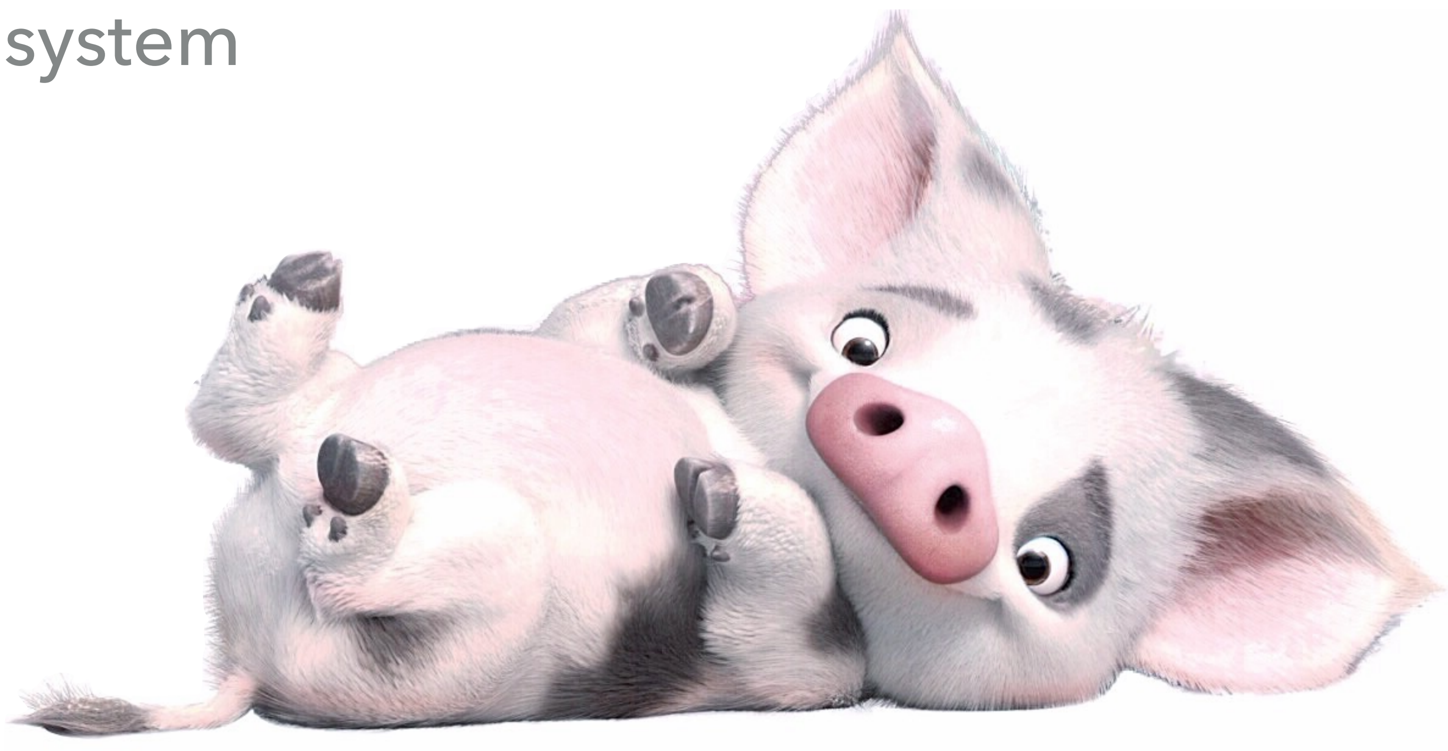
HELP SESSION

ANIMATOR

OUTLINE

- Application interface
- Project requirements
  - Curves: Bezier, B-splines, Catmull-roms
  - Add viscous drag to Emitter Particle system
  - Spring Connected Particle system
  - Cylinder colliders
- Artifact tips!

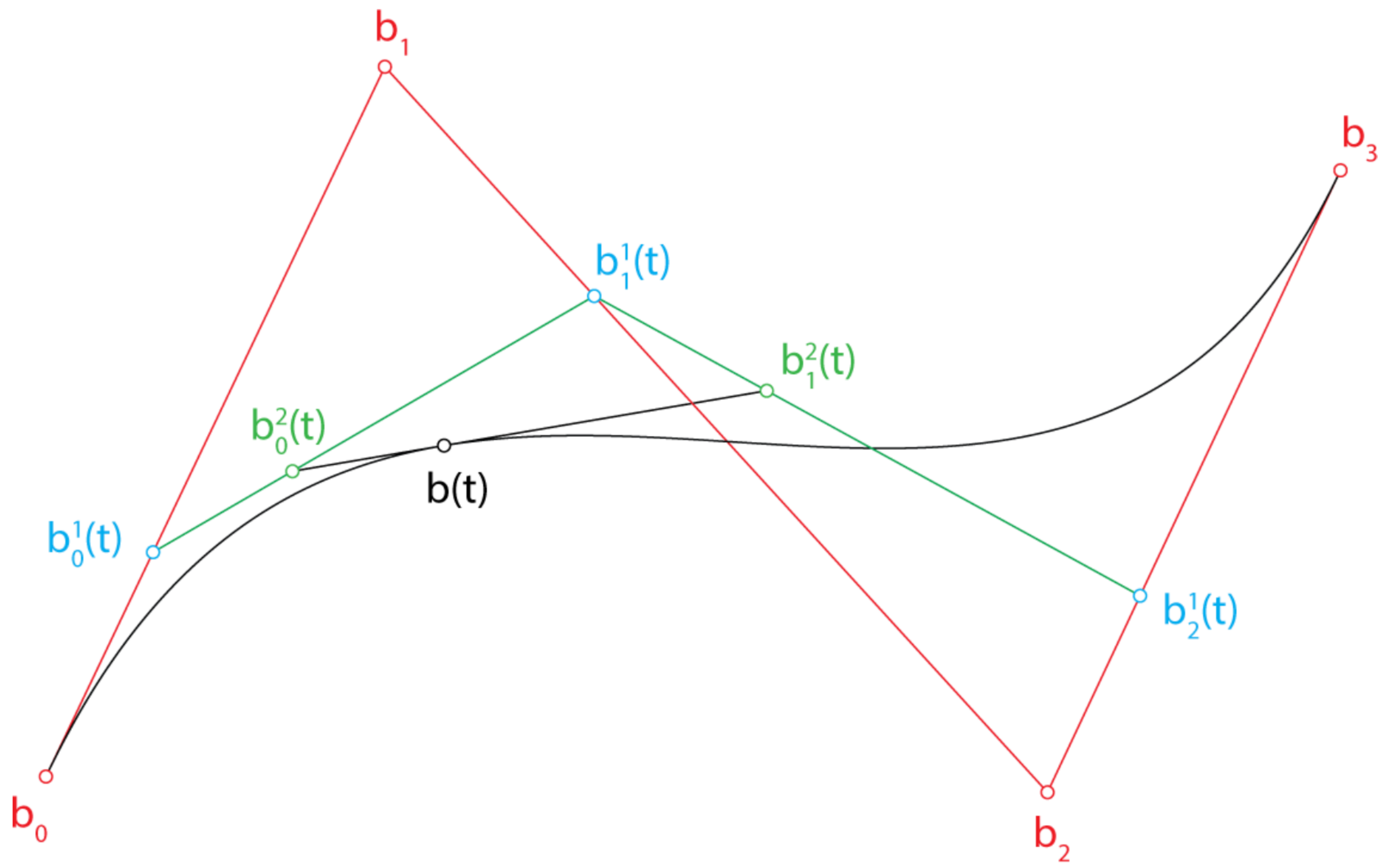


GETTING STARTED

- Clone the Animator skeleton code
  - `git clone git@gitlab.cs.washington.edu:csep557-19sp-animator/YOUR_REPO.git animator`
  - Note: if you want to include any extra credit from Modeler, you'll have to copy or merge that code over
- Note the **Animation** tab in the bottom window
  - Left: Keyable properties for the selected object
  - Right: Graph window
  - Bottom: Time slider
- Interface is represented by **AnimationWidget** - add extra UI here

DEMO

CURVES



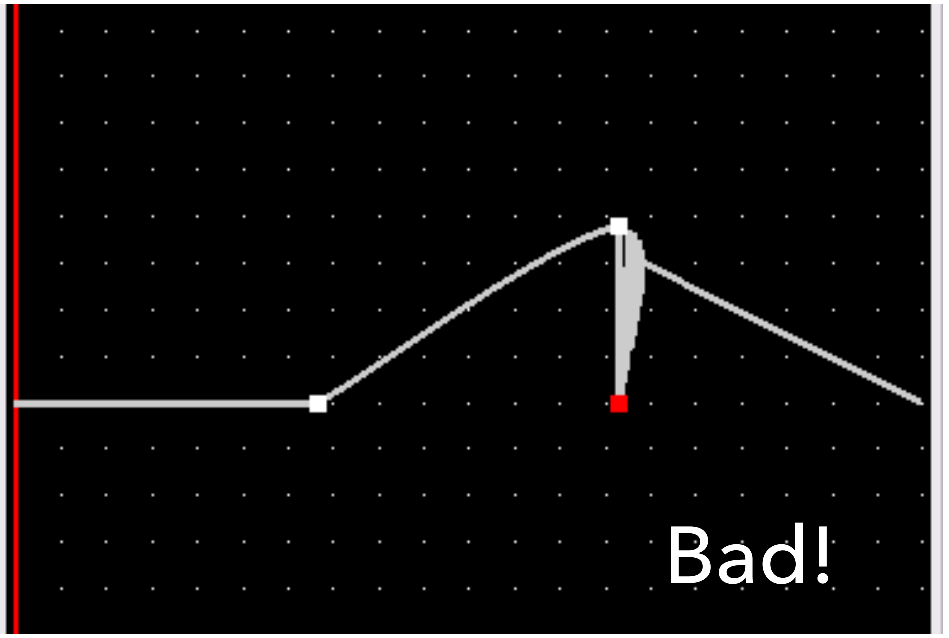
CURVE EVALUATOR

- Implement the evaluateCurve function for each curve
  - **`ctrl_pts`** - a sorted collection of control points that the user specifies in the graph editor
  - **`density`** - how many times to sample between control points
- Note that CurveEvaluator is constructed with:
  - **`max_x`** - animation length in seconds
  - **`wrap_y`** - flag for whether to wrap end to beginning (EC)
- Use the LinearCurveEvaluator code as an example

REQUIRED CURVES

- Bezier
  - Adjacent Bezier curves share endpoints
- Catmull-Rom
  - Interpolate endpoints (double them)
  - Make sure your curve is a function!!
- B-Spline
  - Interpolate endpoints (triple them)

HOW IT WORKS



- Control points are sorted for you
- Your evaluated control points then will also be ordered, so…
  - They must be a function! x should not decrease.
- Evaluation function draws line segments between each of your evaluated points to create a smooth curve
  - Use control points to calculate your evaluated points which draw your curve - should always extend from time 0 to animation_length
  - How might you calculate evaluated points so your curve wraps?

BEZIER CURVES

$$b_0^3(u) = (1-u)^3$$
$$b_1^3(u) = 3u(1-u)^2$$
$$b_2^3(u) = 3u^2(1-u)$$
$$b_3^3(u) = u^3$$

- Use the Bernstein polynomials from lecture
- Use linear interpolation when there are not enough control points (< 4 for a set)
- Base requirement: sample `u` at regular intervals for 0 <= `u` <= 1 (use the `density` parameter)
  - EC: Adaptive subdivision with de Casteljau's algorithm (see website)

CATMULL-ROM CURVES

- C^1 continuity
- Similar to Bezier, but now you evaluate a transformed set of points
- Use linear interpolation when there are not enough control points (< 3 for a set)
- Double your endpoints to interpolate!

B-SPLINE CURVES

- C^2 continuity
- Another transformation on your set of control points (called de Boor points)
- Use linear interpolation when there are not enough control points (< 3 for a set)
- Triple your endpoints to interpolate!

PARTICLE SYSTEMS

EMITTER PARTICLE SYSTEM

- Your first requirement is extending the ParticleSystem class
- Run the skeleton to see how it works
  - Includes constant force (set to gravity as default)
  - Includes sphere and plane collision
    - Go to SceneObject -> Create Collider
  - Uses Euler's method to update position and velocity
- It also includes some extra controls, like changing the Particle mesh, material, scale, initial velocity, etc.

EMITTER PARTICLE SYSTEM - REQUIREMENTS

- Add viscous drag (f = -k_drag * velocity)
  - UI slider is provided
- Add support for cylinder collision
  - CylinderCollider class is already defined, but you have to implement the effect of this collider against the emitter particles system
  - Particles should bounce off both endcaps and the curved body, at the correct normal
  - **Restitution** attenuates the normal component of the reflected velocity
  - The solution does not demonstrate this yet; expect an update in a few days!

NOTE: CALCULATIONS IN WORLD SPACE!

- If you spawn your particles from a node in your hierarchy that isn't the root, it still behaves correctly
- Find the world coordinates for your particles - not local
    - Why? Ex. If we apply gravity in the local coordinates of your particle system, then the force in the -y direction is dependent on the orientation of that node, not the -y of the world
    - Apply the model view matrix (i.e. **model_matrix_**) to your position, velocity, etc. vectors
- This is done for you in ParticleSystem, do the same in ConnectedParticleSystem (your spring system)

FIXED PARTICLE SYSTEM

- Skeleton outline is provided in the ConnectedParticleSystem class
  - Fill in the REQUIREMENT sections to properly run and update the simulation
  - You will need to add member variables and possibly methods to fully implement your system
- What is the difference?
  - This system has a **fixed** number of particles with spring forces that interact **between** the particles
  - Most commonly, this is used to create a mesh where the particles act as vertices
    - Deforming cubes, flexible hair or grass, cloth
    - `glRenderer::Render(SceneObject&, ConnectedParticleSystem)` handles drawing the mesh lines between particles - edit this if you wish to change the rendering
- May reuse parts from ParticleSystem.h or use inheritance; you design it

FIXED PARTICLE SYSTEM – REQUIREMENTS

$$f_1 = -\left[k_{spring}(\|\Delta\mathbf{x}\| - r) + k_{damp}\left(\Delta\mathbf{v}\cdot\Delta\hat{\mathbf{x}}\right)\right]\Delta\hat{\mathbf{x}}$$
$$f_2 = -f_1$$

- Implement spring force using Hooke's law with damping
  - See the lecture slides; note that force gets added to both particles
  - Must also use Euler's method
    - EC: More powerful methods like Runge-kutta
- Apply an additional force
  - Constant (gravity), electromagnetic, buoyant, flocking (probably with sets of connected particles); may earn EC
- Implement collision detection (sphere, plane, cylinder)

TIPS

- Although not required, think about how you may want to extend or apply these particle systems to your animation later
- The sample solution uses springs to implement a deformed cube
  - Note: it connects every possible pair of vertices; more springs = more stable
- Springs, especially stiff ones (or over-damping), get unstable
  - It can be finicky to find the right values
  - The sample solution and **assets/scene/spring_particle_system.yaml** have examples of constants in systems with gravity and without
- Realtime Play mode skips frames, so has unstable Euler integration (this includes collisions)

HOW TO MAKE IT COOLER

- Curves
  - Tension control for Catmull Rom
  - Allow control points to have (or not have) C0, C1, C2 continuity
  - Curve wrapping (UI provided already)
- Particles
  - Cloth simulation
  - Flocking
  - Billboarding (see code comments)
    - And transparent textures -> Fire, snow, leaves
  - Baking
    - Improves performance for complicated simulations with many particles

TIPS FOR GOOD ARTIFACTS

LIGHTS CAMERA ACTION!

HAVE A PLAN

- This artifact takes more time than the others - we give you a week
- Keep it simple, have realistic goals. If you finish early, go back and enhance
- Sketch out storyboards and key poses/frames before implementing
  - Much easier to iterate on paper than in the animator program
- Complicated != better. Well animated simple models are more entertaining than poorly animated complicated models
- Read John Lasseter's article on animation principles!! https://courses.cs.washington.edu/courses/cse457/15sp/projects/animator/linkedItems/lasseter.pdf

TIPS FOR YOUR MODELS

- You may update or add more models as you like
- Many modeler artifacts were not properly "rigged"
  - Fix this now or else you won't be able to animate
  - Ex. body parts have joints. If it bends, use either a sphere node or an empty node.
  - Translate the child to where you'd like it. Now when you rotate the parent (joint), your child node pivots correctly
- A Blinn-Phong shader with texture mapping can add a lot, and is fairly easy to implement
  - Look at the provided texture.frag and texture.vert as reference
  - Find or make your own textures by using checkers.png as a reference for how the texture is mapped on your 3D objects (and then use Paint, GIMP, Photoshop, etc.)
  - Can use transparent textures

CHOICE OF CURVES

- Catmull-Rom is usually the preferred curve choice
  - But unless your project supports the option to add C1 discontinuity at will, you might find yourself fighting the Catmull-Rom to create pauses and control the timing
  - Bezier spline works well for things like animating a bouncing ball
- Note on keyframing:
  - Auto-keyframe is a mode (turned off by default) that creates keys whenever a transform is changed
  - Otherwise, skipping the time without 'keying', will erase the transform change!

IMPORTANT COMPOSITIONAL COMPONENTS

- Timing
  - Consider timing and shot planning before getting specific about joint rotations or positions
  - Total length **MUST** be < 60sec. We recommend 24 or 30 fps.
- SFX + Music
  - Greatly enhances cohesion of your artifact
  - If your idea includes a theme or stylization, very effective to time your animation with events in the theme music
- Lighting
  - Like sound, super important compositionally - can signal story and mood
- Camera Angle
  - Changing perspective between two shots or panning/zooming camera can add depth
  - Do not go overboard! And remember the *180 degree rule*.

PUTTING IT TOGETHER

- Make sure you keep your original model .yaml file separate
- We recommend breaking up your intended artifact into shorter clips or "shots" and combining them in the end
  - Can incrementally complete your artifact
  - Save a new .yaml file for each shot, and build off the base of your original model (or from your last shot)
- **SaveAs often** - there are no undos
- Your animation is saved in frames, and you must composite
  - Blender is free, and we provide a tutorial
  - Adobe After Effects and Premiere can also composite your frames into a movie - and much more easily too
  - < 60s, and **must be H.264 mp4 format**

THE END

GOOD LUCK