CS52 Assignment 3: Animation

Out: May 2. Due: May 18.

Introduction

In your third assignment, you will write code to produce simple animations that will play back using interactive drawing through OpenGL. The setup of this assignment is the same as the modeling one.

The final result of this assignment is an interactive program with support for the following features.

- Keyframe Animation
  - Keyframed Bezier Splines
- Patricle System
  - Point source
  - Age-based particle deletion
  - Dyanmics based on simple constant forces
- Skinning
  - Skin deformation based on an array of bones
  - Automatic weight computation

To ease your debugging, we are providing the compiled version of the solution code, which can be used to compare your results with ours. Do not try to decompile the code.

Requirements

1. Implement the `Tranform.animate` function that updates the values of the transform properties `translation`, `rotation` and `scale`, when the corresponding variation is present, by calling the variation evaluation function. Also Restore the state of a transform to the original at time 0 by `Transform.restartAnimation`. Remember to propagate to the children.

2. Implement the `KeyframeBezierSpline.evaluate` that evaluates the keyframed Bezier spline at time t.

   This type of splines is similar to our previous Bezier curve except that its segments' domains are defined by the keyframes. In particular the segment k of the spline

corresponds to the t interval $[t_k, t_{k+1})$ where t_k are the keyframe times stored in the `keyframeTimes` array. Within this interval, the value $s = (t - t_k)/(t_{k+1} - t_k)$ can be used to evaluate the spline segment defined by

$$\mathbf{p}(s) = b_0(s)\mathbf{p}_{4\cdot k} + b_1(s)\mathbf{p}_{4\cdot k+1} + b_2(s)\mathbf{p}_{4\cdot k+2} + b_3(s)\mathbf{p}_{4\cdot k+3}$$

where $\mathbf{p}_i$ are the control points stored in `keyframeControlPoints`.

In order to implement this function, you should first find which k segment the current t belongs to (k_i is sorted), then compute s and finally evaluate the point.

3. Implement the `ParticleSystem.animate` function that creates, deletes and update particle state based on their age and dynamics.

   The particle states are to be stored in the `particles` array. When first calling the `animate` function, if the particles do not exist yet, you should allocate an array of `numParticles` and initialize their state by calling the `source.createParticle` for each value.

   When calling `animate` with an already created particle array, you should first check if a particle is to be kept by calling `isParticleDead`. If the particle should be discarded, simply create a new one and call `source.createParticle` on it. Otherwise, update the particle state by calling `updateParticleState`.

4. Implement the `ParticleSystem.isParticleDead` method to determine if a particle is to be eliminated. In our system, particles are only eliminated based on their age. In particular a particle with an age smaller than the `ageMin` will never be deleted, while if the age reaches `ageMax` the particle is to be deleted. When the particle age is between the min and max, we will just decide randomly if the particle need to die based on

   $$dead = (age - ageMin)/(ageMax - ageMin) > randomValue$$

5. Implement the `ParticlePointSource.createParticle` which sets the particle state as originating from a point source (like a fountain for example). Particles will have their location in the source location whole their velocity will have random direction but with a magnitude chosen randomly between `velocityMin` and `velocityMax`. To make things a bit prettier, we will set the color of a new particle as the sum of the source `color` and a random color color that has each channel between 0 and the corresponding value in `colorVariation`.

6. We will finally update the particle dynamics in `ParticleSystem.updateParticleState` by first computing the total force acting on each particle and then using the Euler equation to integrate the particle velocity and position.

7. You will finally implement skinning by first determining automatically the vertex weight for a given mesh and then transforming all the points for each frame. Skinning is implemented in the `SkinnedMesh.computeVertexWeights` method.

   This class represent skinning as a tessellated skin mesh with an attached set of bones. The skin mesh and the bones are defined in the same space at $t = 0$. Each bone is defined by a translation and a rotation (with optional animations) together with a size that represents the bone object (as a box) as well as the bone influence used

later. The bone object in world space is the transformed box originating from the box $(-1, 0, -1) \times (1, 1, 1)$ in its local space.

8. Implement the function `Bone.animate` that updates the translation and rotation of the bone object and stores the transformation in the `m` matrix (and its inverse transpose in the `nm` matrix). We will define M as

$$M = T_{\mathbf{t}} \cdot R^x_{\theta_x} \cdot R^y_{\theta_y} \cdot R^z_{\theta_z} \cdot S_{\mathbf{s}}$$

   and its inverse transpose can be computed from the `Mat4.normalTransform`.

   You should also implement the function `Bone.computePoseTransform` that evaluates the inverse of the two matrices at time 0. These are the inverse transforms for points and normals in pose space. Note that you can compute this transform without actually inverting any matrix.

9. To compute the weights, we will treat each bone b as a capsule and compute the point distances d_k from it. Once these are known we will set the unnormalized weights as d if $d_k < d_M$ or 0 otherwise, where d_M is the maximum distance at which the bone has an effect (stored in `autoWeightsMaxDistance`). The weights are then normalized by their sum. If the sum is 0, simply pick the weight with smallest distance to be 1 and everything else 0.

   To compute the distance d from a bone, we first transform the point in the bone local coordinates $\mathbf{P}_l = M_0^{-1} P$. Then if $P_l.y$ is between 0 and 1, we simply take $d = \sqrt{s_x^2 P_l.x^2 + s_z^2 P_l.z^2}$, where $\mathbf{s}$ is the size of the capsule. If $P_l.y < 0$ we will take $d = \sqrt{s_x^2 P_l.x^2 + s_z^2 P_l.z^2 + s_y^2 P_l.y^2}$ and finally for $P_l.y > 1$ we will set $d = \sqrt{s_x^2 P_l.x^2 + s_z^2 P_l.z^2 + s_y^2 (P_l.y - 1)^2}$.

10. Finally, we will transform each point and normal by first evaluating the bone transforms (in the `Bone.animate` function) and then computing the weighted averages. For the point this is

$$\mathbf{P}_i(t) = \sum_b w_{ib} M_b(t) M_b^{-1}(0) \mathbf{P}_i(0)$$

Framework Overview

We suggest you use our framework to create your program. We have removed from the code all the function implementations you will need to provide, but we have left the function declarations which can aid you in planning your solution.

Following is a brief description of the classes provided in the framework that are substantially different from the previous homework. We have taken various steps to simply the framework code as much as possible. In particular, you will notice that most of the variables are declared as public and all classes are in the same package. Please note that this is not good programming practice! We have done this to help making the code more readable by reducing the length of the classes.

GLRenderPanel. We have added a new drawing mode for bones and well as the option to draw particles as points are geometry. There are also controls to start/stop/step-trough/reset the animation and change its timestep in millisecond.

SkinnedMesh This is a new surface type that implement mesh skinning. Currently it is the only Surface for which the `animate` function does any work since this surface is deformed over time.

Submission

Please send your code and compiled class files to fabio.pellacini.sapienza@gmail.com. We will run your code by calling the main method provided. Also add images by screen-capturing the program at time $t = 1$.

Extra credit

Choose *some* of these items as extra credit.

1. Update the particle system dynamics to support collisions with simple objects (sphere and planes). You can use the same math as the ray-object intersection you know from your raytracer to compute the position and normal of the intersection. You can then update the velocity and position of the particle after intersection. You are welcome to grossly approximate this as soon as it looks reasonable for small time steps. Asks us for help to better define the problem if needed.

2. Create new particle systems sources and forces to simulate snow, rain and wind. Remember that simple approximations work really well. Ask us for help to better define the problem if needed.

3. Implement proper skinning by representing the bones as hierarchy instead of a list. This way character motion is strongly facilitated. Talk to me if you want some motion capture data to try this on.