Programming Assignment 6: Curves, Surfaces, and VR
Fall 2016

Submission deadline: Thursday, December 22, 12 pm (noon)

In this assignment you will first implement surfaces of revolution[1] using Beziér curves. For the rest of the assignment you have the choice between two options: you will either use your surfaces of revolution to create a complete scene, and also implement a surface subdivision algorithm. Alternatively, you will implement a small VR application. As usual, the assignment must be submitted on Ilias before the submission deadline on Thursday, December 22, 12 pm (noon) and the assignment must be presented to one of the teaching assistants.

While task 1 is mandatory, you have the choice of either solving in addition tasks 2 and 3 (surface subdivision and scene modeling) or task 4 (VR application). Note that you will get points for **either** task 2 and 3 **or** task 4, but not for all of them. So you need to decide in advance which task you want to solve and turn in. If you decide to solve task 4, please sign up on Ilias to reserve time slots to work on one of the VR devices in the INF lab (see details below).

In addition, you can solve some bonus assignments as described in task 5 to earn up to halve a grade point on the final exam (e.g. if your points in the exam correspond to a grade of 5 you would get a 5.5 instead).

1 Bodies of Revolutions with Bezier Curves (4 Points)

Implement a function or class to generate surfaces of revolution using Bézier curves. Surfaces of revolution are constructed by first defining a 2D curve on a plane. The curve is then rotated around an axis on that plane, thus generating a surface. The idea of this task is to use piecewise cubic Bézier curves to define surfaces of revolution. Further details and explanations concerning surfaces of revolution can be found at Mathworld or Wikipedia.

To create a triangle mesh, the curve first has to be evaluated at a sequence of points in the plane. These points can then be rotated around the rotation axis and joined to a triangle mesh.

Furthermore, define surface normals and texture coordinates for all vertices. Normals can be computed as follows: Assuming your 2D curve is of the form $(x(t), y(t), 0)$. You can first compute the tangent $(x'(t), y'(t), 0)$ of the curve. The matching normal is then

[1]https://en.wikipedia.org/wiki/Surface_of_revolution

$(-y'(t), x'(t))$. The normal is then rotated along with the computed vertices around the rotation axis. To compute uv texture coordinates, you can use the parameter of the curve as u and deduce v from the rotation angle.

Your function for generating surfaces of revolution should have the following inputs and outputs:

Inputs:

- Count n Bézier segments
- Array of the Bézier control points in the xy plane, meaning points $(x_i, y_i, 0)$. To create n cubic segments you need $(n-1) \times 3 + 4$ control points.
- Number of points which should be evaluated along the curve
- Number of rotation steps used for construction

Output:

- Array of vertices
- Array of surface normals
- Array of texture coordinates
- Index array for the triangle vertices

2 Surface Subdivision (4 Points)

Implement the Loop surface subdivision algorithm discussed in the lecture. The subdivision should be applied on a winged edge structure. Note that the class *MeshData.java* already provides a method *createMesh(VertexData)* to create a winged edge structure from a *VertexData* object. The method generates three tables, a *VertexTable*, a *FaceTable* and an *EdgeTable*. The class *MeshData.java* also provides helper methods for using the winged edge structure to find neighboring edges and vertices. For more details concerning these methods please consult the method documentation directly in the code.

Optimally, you implement a method *Loop()* in *MeshData.java*, which from an old winged edge structure generates a list of vertices of a new, subdivided mesh, as well as an integer-array containing information about how to connect the vertices to triangles (similarly as the integer-array in *VertexData*). Using the existing method *createMesh(List<Vertex>,int[])* you can then create a new winged edge structure and a new *VertexData* object. The newly created winged edge structure can then be used for further iterations of the subdivision, while the newly created *VertexData* object can directly be used to render the mesh. To perform multiple iterations of the subdivision it is sufficient to then call *Loop()* multiple times. Demonstrate this functionality by performing (and rendering) one more subdivision of a mesh each time you press a button.

Note that only closed triangle meshes can be transformed to a winged edge structure, and each edge in the mesh needs to correspond to exactly two faces, meaning that the mesh must not have borders. Two simple sample meshes that satisfy these conditions are provided in *Meshes.txt*.

3 Modelling a Scene (2 Points)

Model a scene that includes several surfaces of rotation and/or subdivision surfaces. The scene must contain at least three different objects. You can for example create a still life, which is comprised of a round table with some objects on it. You could for example model a wine bottle, a candle, chess figures or a vase etc. Use suitable textures, material properties and colors.

4 Interactions in a VR Application (6 Points)

In this (optional) part of the assignment you will implement an interactive application for a VR device (HTC Vive). The goal is to create a simple squash game. More specifically the game should support throwing a ball that bounces off walls and can be hit by a racket.

Basecode: We provide code to render on the head mounted VR display (HMD) and to access the poses of the HMD and the VR hand controllers. The example scene *simple_VR* already provides such functionality and consists of a box surrounding the player, a ball, and two simple objects representing the controllers. One controller is a small box representing a hand, and one is a rectangle representing a racket. The pose of the HMD controls the camera matrix, while the poses of the controllers determine the transformation matrices of the objects representing them. Your task is now to implement ways to interact with the ball according to the detailed description below.

We provide two new classes, called *VRRenderPanel* and *VRRenderContext*, that you can build on. These classes implement the communication with the VR devices (we are using the OpenVR[2] API). Their functionality is similar to the corresponding *GL* classes. However, *VRRenderPanel* has some additional methods: with *getTriggerTouched(controllerIdx)* resp. *getSideTouched(controllerIdx)* one can query whether the trigger button on the back of the controller with index *controllerIdx* is triggered resp. whether one of the two side buttons of the controller with index *controllerIdx* is triggered. Furthermore, one can trigger a haptic feedback (i.e. a short vibration) with *getTriggerTouched(controllerIdx, strength)*.

Assignment: Your application should provide the following functionality:

- Pick up the ball and throw it with the "hand" controller. The ball is picked up when the "hand" touches it (or is inside it) and the trigger button is pressed. As long as the ball is picked up, you should be able to drag the ball around following the movement of the hand controller. Also, activate the haptic feedback of the hand controller as long as the ball is being dragged. Note that the movement of the ball should be smooth, that is, when picking it up it should not suddenly jump to the position of the

[2] https://github.com/ValveSoftware/openvr

hand. When the trigger button is released while dragging the ball, the ball should be thrown into the direction in which the hand was moving when the trigger was released, with a speed corresponding to the controller movement. One simple way to determine this direction and speed is to compare the transformation matrix of the hand at the moment of the trigger release with the transformation in the previous frame. You do not need to consider the spin of the ball.

- When throwing a ball, gravity should influence the trajectory of the ball. The simplest way to implement this is by applying a translation towards the ground at every frame in addition to the throwing direction.
- The thrown ball should bounce off the walls. The bouncing must not be implemented in a physically correct way, approximating the bouncing is enough. One way to do so is to simply reflect the current moving direction of the ball on the normal of the wall that was hit. This requires a method to detect intersections of the ball with the walls. We recommend the following approach:
  1. Check if there is an intersection of the ball with one of the walls (i.e. perform a sphere-plane intersection).
  2. If so, reflect the current motion direction of the ball on the normal of the wall. Note that the normals of the walls must point inwards.
  3. Approximate some loss of energy due to the bouncing by slightly reducing the length of the new (reflected) motion vector.
- It must be possible to hit the thrown ball with the "racket". Hence you need a method that checks if the ball hits the racket and a way to compute the new direction of the ball. If the racket hits the ball, trigger a short haptic feedback on the racket controller. We recommend the following approach:
  1. Compute the intersection position of the ball with the racket. If the ball is a sphere and the racket a rectangle, then the simplest way to check for intersection is to use a sphere to axis-aligned box intersection approach, by transforming the sphere into the racket's own coordinate system (see *sphere_box_intersection.pdf* Sec. 16.13.2 in the additional material on Ilias).
  2. If there is an intersection, reflect the movement direction of the ball on the normal direction of the point on the racket that was hit. Also add a short haptic feedback when this happens.
  3. The new (reflected) direction and speed of the ball must be summed up with the direction and speed of the intersected point on the racket. The direction and speed of the intersected point on the racket can be determined by comparing its position in the frame when the intersection happens with its position in the frame before.

Code Installation: We updated the *git* repository with the additional code for the VR application. You can either try to update your current project by right clicking on the package explorer and selecting *Team→Pull*. This may require some conflict resolving. If you are not familiar with *git* then we recommend that you create a new workspace and re-import the whole project from the *github* repository (follow the instructions from Assignment 1). No matter which avenue you choose, you will still need to add the new *openvr* project as dependency to jrtr. To do so in eclipse, right-click on the *jrtr* project in the package explorer, select *Properties→Java Build Path→Projects→Add...* Then add the *openvr* project as dependency and apply the changes.

Running the code: Our code uses *OpenVR* and requires *SteamVR* to run. Specifically, the *SteamVR*-panel must be open when running the application. *SteamVR* is installed as part of Steam and can be run directly from Steam (click on the "VR" symbol on the upper right of the Steam window). In *SteamVR* you can check if all devices (that is the HMD, the two controllers and the two sensors) are currently tracked. If they are not, running *simpleVR* project will throw an exception. Note that a bug in the current vesion of *OpenVR* can cause that the pressing of buttons will not be registered anymore. If this happens, reboot the VR device (in the *SteamVR* panel click on *SteamVR→Devices→Reboot Vive headset* and wait for *SteamVR* to restart).

Important: The VR headsets to develop and test your application are available in the INF building in room 214 (INF building, Neubrückstrasse 10, map). We have set up two HTC Vive devices with workstations for that purpose. Because of the limited number of VR headsets you should solve this task in groups of 2-3 people. To simplify organization, each team must announce on Ilias who is in the team and when you want to work with the VR devices in the lab. We will announce time slots on Ilias so each team can allocate some time to work on the task. If you own a HTC Vive yourself you are of course free to solve this task at home. However, you should then still make sure that you can demonstrate the assignment on one of the machines in the lab.

5 Bonus Task

The solution of this task will be awarded with a bonus of up to 0.5 grades towards the final exam. Implement one or multiple of the following algorithms:

- Shadow mapping. It should be possible to interactively move the light source. Also use Percentage Closer Filtering.
- Bump mapping. It should be possible to interactively move the light source. First test using a plane. For arbitrary objects, it is more complicated. You can for example use xNormal to compute the tangent vectors for a triangle mesh with texture coordinates.
- Reflection and refraction with environment maps. Use Schlick's approximation for the Fresnel equations.
- Irradiance Environment Maps. Use HDRShop to generate the irradiance environment maps.
- Ambient Occlusion. Use the xNormal Tool to prepare triangle meshes with appropriate data.

- Catmull-Clark Surface Subdivision.
- Extend the VR application from task 4. You could for instance make the simulation of the ball movement more realistic by including the spin and center of mass in the computation. Alternatively you could extend the game by a simple GUI interface that is visible in the HMD that counts the score (like how often did the ball bounce on a wall before hitting the ground) and/or some properties (like for instance the velocity of the ball). You could also try to add sound effects to the application that are triggered when the ball bounces off a wall or is hit by the racket.

Demonstrate your implementation using a scene as appealing as possible. The grading of this task will consider technical difficulty as well as the aesthetic impression and the effort put into the construction of the scene or the VR application.