Introduction to Computer Graphics
Project Part 3 – Interaction & Animation

Minh Dang, Andrea Tagliasacchi, Mario Deuss

Handout date: Tue May 6th
Submission deadline: Wed May 28 (8pm)

Late submissions are not accepted

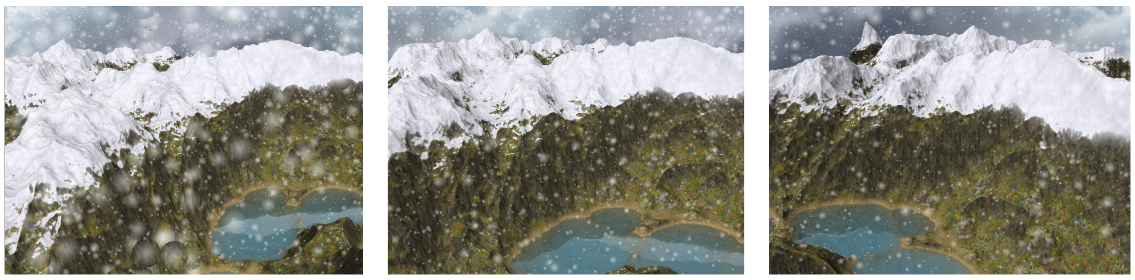
Figure 1: Frames from a camera path through snowy mountains

1 Basic camera control

In the framework, we have provided you with a trackball (see *glfw_trackball.h*) which allows to translate, rotate and scale the model. Another method to navigate the terrain you have generated in previous stages is to change the camera while keeping the model fixed. In this section, you are asked to implement basic camera controls. To start with, you can check *glfw_trackball.h* to see how the trackball is implemented.

1.1 Fly through mode

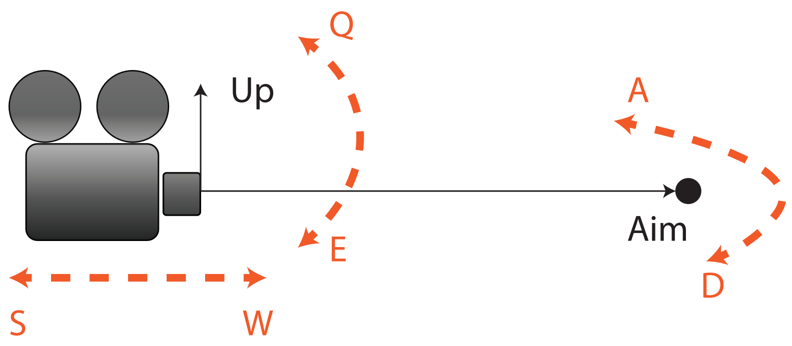

Figure 2: Camera controls

The perspective camera used in the framework has three components: (i) the camera position `cam_pos`, (ii) the aim (or look-at point) `cam_look` towards which the camera is pointing, and (iii) the up direction `cam_up`. From these components, `Eigen::lookAt(cam_pos, cam_look, cam_up)` generates for you the view matrix of the camera.

You are asked to implement a fly-through camera mode by mapping certain keys to change the position, the aim and the up direction of the camera. To be specific, use *W* and *S* to move the camera along the view direction, *A* and *D* to rotate the aim about the up direction and *Q* and *E* to roll the camera up and down (See Fig. 2.)

Add inertia to your camera controls. For example, after pressing *W*, the camera velocity increases by a certain amount and then gradually decreases over time to 0. Similar behaviors are expected for rotating and rolling the camera.

See `http://lgg.epfl.ch/teaching/icg/fly_over.mp4` for an example of the fly-through mode.

1.2 FPS exploration mode

Another interesting way to navigate your terrain is to control x and y components of your camera position, while snapping the camera to the corresponding height of your terrain. This is similar to the classical *first-person-shooting (FPS)* navigation mode in computer games. You can use `glGetTexImage(...)` or `glReadPixels(...)` to access the generated height map from FBO. Use the same keys as in the previous section to control your camera.

1.3 Advanced - Physically realistic movements

Objects don't accelerate instantaneously as they have a mass. Build a simple physical system to emulate a physically plausible motion (and possibly integrate jumping).

2 Camera path control

In this portion of the project you will develop the necessary algorithms to have the camera path automatically move along a *user-defined* trajectory. A video screen-capture of this is available here: `http://lgg.epfl.ch/teaching/icg/camerapath.mp4`, and is is the result of a single camera sweep along the simple bezier curve illustrated in the figure below.

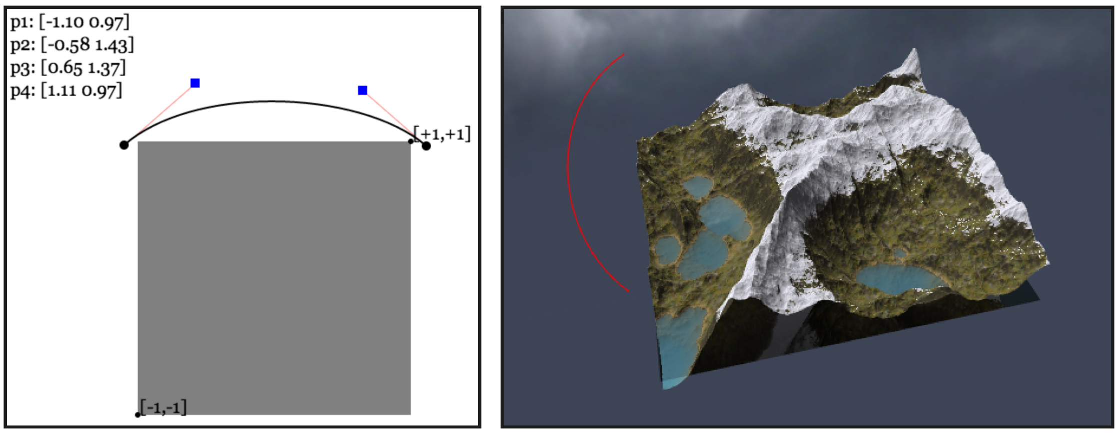

Figure 3: left) A simple 2D javascript interface that allows you to edit the control points of a cubic bezier curve (`http://lgg.epfl.ch/teaching/icg/bezier.html`). right) an example of the camera path (where $y = .5$ for all control points) rendered as a GL_LINE_STRIP.

2.1 Computing and rendering a Bezier curve

The camera trajectory will be modeled with a (piecewise) *cubic Bezier* curve. You could start by using the points in Figure 3-(left). However, for your final results, we encourage you to design a camera path that best illustrates the techniques you developed.

Given the control points and the parameter α, you need to evaluate the 3D position $\mathbf{x}(\alpha) = (x(\alpha), y(\alpha), z(\alpha))$ for $\alpha \in [0, 1]$. To achieve this task you have two different options. The first method (very simple) is to *directly evaluate* the cubic polynomial for the given parameter α. Another approach (more advanced) is to use the computationally efficient "de Casteljau" algorithm.

To make sure your bezier implementation is correct, we **strongly** encourage you to render the camera path in the scene; see Figure 3-right.

Advanced tasks

- Compute the curve using the "de Casteljau" **subdivision algorithm**[1]. To begin, use a fixed number of recursions (we used 5 splits) and perform the split at $\alpha = 0.5$; note that, after subdivision, a piecewise linear approximation of the path is obtained by simply concatenating the control points.
- On top of the camera path also render a **pictorial representation of the camera** and animate it without changing the (OpenGL) camera position. To know exactly what we are referring to, please see the *4m50s* fragment of this video: `https://www.youtube.com/watch?v=vasfFsxCsAE`.
- Rather than using a single bezier curve (4-points) construct a complex path as a **concatenation of Bezier curves**. Given two concatenated control polygons $\mathcal{P}_A, \mathcal{P}_B$ remember to consider the relationship in between p_3^A and p_2^B as explained in the textbook.
- Instead of setting the control points at compile time, integrate the camera controls you defined in Section 1 to be able to **interactively design a camera path**. Use the camera position to specify interpolatory constraints of each control polygon (p_1, p_4) and devise a way to define (p_2, p_3) of each segment.
- Instead of using a fixed (user defined) recursion depth for "De Casteljau" implement a control scheme for **recursion termination**. The basic idea is that when the control polygon is flat then it can be trivially represented by a line segment, thus no further split is necessary. Consequently, the problem becomes to devise an (efficient) way to measure the *flatness* of the convex hull. Please refer to this technical report for more details: `http://algorithmist.net/docs/subdivision.pdf`

2.2 Animating the camera

In the previous section we generated a Bezier curve, now let us proceed to link this curve to the camera. Recall from Section 1 that the camera is fully determined by the triplet of vectors $(\mathbf{c}_{pos}, \mathbf{c}_{look}, \mathbf{c}_{up})$. Note that we made the simplifying assumption[2] that the camera has a trivial up-vector $\mathbf{c}_{up} = [0, 1, 0]$ as well as a fixed look-at position $\mathbf{c}_{look} = [0, 0, 0]$.

If we want to move the camera along the bezier curve (starting from $t = 0$) over a time span of 10 seconds a possible (erroneous) approach would be to set the camera position as $\mathbf{c}_{pos}(t) = \mathbf{x}(10\alpha)$. This is erroneous because Bezier curves **do not** possess a natural *arc-length parameterization*. This implies that the velocity of your camera along the path **will not be constant**.

[1]Angel, Shreiner, "Interactive Computer Graphics (6th ed.)", Section 10.9.2

[2]Notice these assumptions were made because our terrain is parameterized in the $x - z$ plane and spans $[-1, 1]^2$

To address this issue a simple solution is to sample/evaluate the Bezier curve by uniform sampling $\tilde{\alpha} = [0 : \delta_\alpha : 1]$ and then approximate the curve by a line strip[3]. Evaluating the piecewise linearized curve $\tilde{\mathbf{x}}(\alpha)$ then simply amounts to seek the point at a distance $\alpha \cdot \ell$ from $\tilde{\mathbf{x}}(0)$, where distances are measured **on the curve** and ℓ is the total length of the line strip. Our camera position will then be $\mathbf{c}_{pos}(t) = \tilde{\mathbf{x}}(10\alpha)$.

Advanced tasks

- Rather than using a fixed look at position $\mathbf{c}_{look}$ use a second bezier curve to dynamically control the look-at position.
- Similarly to what we saw in Section 1, instantaneous changes in velocity are not natural. Furthermore, artists like to have **direct control** on the velocity of the camera along a path. Using a Bezier curve, create a function $f : [0,1] \rightarrow [0,1]$ to remap the $\alpha = [0,1]$ parameterization domain. Using this approach devise a way to obtain the so called *ease-in/out* effect for camera movement.

3 Advanced - Particle System

Particle system have been used extensively to visually simulate various effects such as fire, stars, liquids, clouds, dust. For our landscape, we propose you to use one for snow modeling, e.g. as in the screen cast: `http://lgg.epfl.ch/teaching/icg/Snowing.mov`.

A particle system is usually modeled as a set of points in space together with some generators. The points move due to physical or non-physical laws, depending on what they try to model. In our case we opt for a very simple model with a constant acceleration pulling downwards plus some random offsets to synthesize details: Let p_t^i and v_t^i be the position and velocity of the i-th particle at time t. Given an acceleration a, the position p_{t+1}^i and velocity v_{t+1}^i can be estimated by explicit Euler integration:

$$v_{t+1}^i = v_t^i + \Delta_t(a + r), \tag{1}$$

$$p_{t+1}^i = p_t^i + \Delta_t v_t^i, \tag{2}$$

where Δ_t is the time passed between t and $t+1$ and r is a random vector. In our implementation of snow we reset/delete particles that fall below the water level by setting their position to a random point of a quad above the terrain, and their velocity to zero. In our fragment shader we use a radial gradient to draw the particle using `gl_PointCoord`, and `discard` to reject fragments outside of a circle. We also rescale the particle sizes in the vertex shader by `gl_PointSize` by the inverse of squared distance to the camera, which you have to enable by `glEnable(GL_PROGRAM_POINT_SIZE)`.

Advanced task: Particles on the CPU There are various ways of implementing this in OpenGL: You can keep the positions and velocities on the CPU, perform the integration step there as well. Then send the particles positions to the GPU using `glBufferSubData`, which, in comparison to `glBufferData` only uploads without re-allocating memory. Also consider using `GL_DYNAMIC_DRAW` when allocating your buffer to get better performance. For drawing points use `GL_POINTS`.

Advanced task: Particles on the CPU Integrating the positions and velocities on the CPU and sending them to the GPU each and every time will not scale well with the number of particles. For bigger systems you can represent the particle positions and velocities each in a texture. The

[3]... and this is why: "The arclength of a Bezier curve can be very complicated; for a quadratic Bezier you have a complicated expression involving logarithms/inverse hyperbolic functions, and for a cubic Bezier one now requires elliptic integrals." `http://math.stackexchange.com/questions/12186/arc-length-of-bezier-curves`

integration can then be done on the GPU in the fragment shader while rendering a full-screen quad to a second texture of positions and velocities. Also, after initialization the data never needs to be transferred between GPU and CPU. To get a texture with 3 unbound floats per pixel use `GL_RGB32F` as internal format.

Advanced task: Extending the Particle System We encourage you to change the particles system in any way you like. Here are some inspirations: There are many other phenomena that can be modeled by particles, e.g. rain, clouds, pollen or dust, stars, vulcanos. Also the possibilities of drawing the particles on the screen can range from simple points to billboards and alpha-transparency, see here: `http://www.opengl-tutorial.org/intermediate-tutorials/billboards-particles/`. Particle systems where particles interact with each other can model fluids, soft bodies and flocking (herd behavior of animals such as birds and fish), but are considerably more complex to implement and expensive to run due to neighborhood queries.