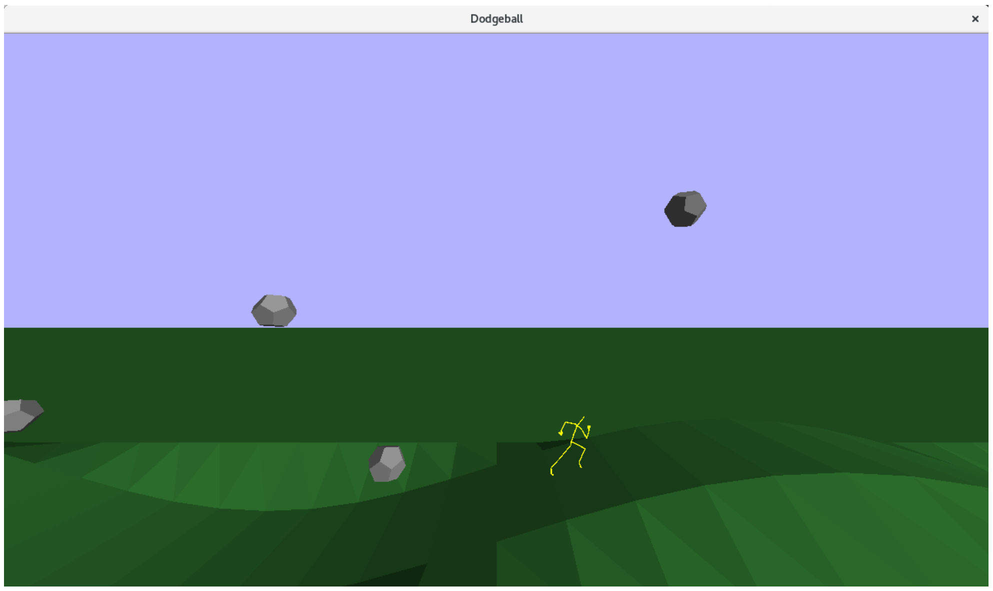
COMP 5893M Modelling and Animation A2: Dodecahedra of Doom

Professor Hamish Carr

Due January 18, 2025 10:00 am



Overview

This assignment is worth 50% of the overall module mark, and is intended to test your comprehension and ability to implement the animation material. You will implement two forms of animation: animation cycles for a character, and physics for a set of bouncing dodecahedra, with collision detection to allow you count how many times the dodecahedra hit the character.

Time

The time budget for this assignment is 50 hours. It will be due at the end of Week 14 (i.e. in the January examination period).

Coding Standards

All code submitted must compile with the stages of compilation in the readme.txt file accompanying the code handout, at the first attempt, on the University's Red Hat lab machines. Students will normally be given exactly one chance to fix compilation issues, but this is not guaranteed.
All code submitted must be written by the student themselves, with no outside assistance of any kind, unless written approval is obtained from the instructor in advance. Students are permitted to use the code handed out, and the following libraries: OpenGL, OpenMP, Qt and the C++ STL.
Students are expected to apply professional coding standards, including meaningful variable names, well-structured OO classes, and meaningful comments. Poor coding practices will attract a penalty of up to 20 marks out of the 50 available.

Starting Code

We have provided a simple OpenGL-based window that displays several different landscapes, which you can switch between with key-based controls. It is recommended that you start with the flat landscape for debugging purposes, then use the other two landscapes later on.

1 User Interface Controls

In order to make it feasible to mark, you MUST use our choice of controls, even if you think you have a better solution. The following keys are defined:

WS	Move the character forwards or backwards
L	Toggle terrain type between flat, stripe and rolling
M	Toggle model between sphere and dodecahedron
Space	Start / stop running by blending to/from rest pose.

2 Animation Cycles [25 marks]

We have provided code for reading in .bvh files - one of the common animation formats, along with several animation cycles sharing a common skeleton (or "armature"). We have not provided a surface model, as it is customary to animate *only* the skeleton in developing animation routines.

2.1 Task Ia: Rendering the Hierarchy [8 marks]

Before animating the skeleton, you will need to draw it directly. A BVH file stores the hierarchy first, followed by the data needed for animation. Instead of storing bones, they store joints. However, since the file specifies the child joints for each bone, this implicitly describes a bone from the origin (offset) of the parent to that of the child. Since the child's offset is described in the parent's coordinate system, all that is needed is to loop through the joints, rendering a cylinder from that location to that of the child.
To do this, you will need to find the offset of each joint, which is stored in the joint_offset field of the Joint class, and redundantly in the boneTranslations array.

2.2 Task Ib: Running Forward [10 marks]

Once you can render the hierarchy, the next task is to use the data in the run cycle to animate the character. The animation data in the BVH file consists of a set of frames: in each frame are a set of "channels", which store floating point values for position and rotation. Although quaternions are a better solution, BVH files store Euler angles, and the rotations specify degrees around Z, Y, X in that order. Remember, however, that this means that we want:

$$\begin{aligned}\mathbf{p} &= \mathbf{R_X}\left(\mathbf{R_Y}\left(\mathbf{R_Z}\mathbf{q}\right)\right) \\ &= \left(\mathbf{R_X}\mathbf{R_Y}\mathbf{R_Z}\right)\mathbf{q}\end{aligned}$$

You can retrieve the pose angles for a given frame from the boneRotations vector, indexing on the frame then the joint ID.
The animations are provided at 24 frames per second. To simplify life, the animation timer has been adjusted to fire 24 times per second.
Animation data normally does not include translation, so you will find that the character is running in place. You will need to add a suitable amount of translation (i.e. forward movement) for each frame. This location information should be kept separate from the BVH data.
Once you start using the curved terrain surfaces, you will have to adjust the vertical position of the character as well based on the x,y position. You can call Terrain::getHeight() to get the correct z value for a given x,y.

2.3 Task Ic: Starting & Stopping [7 marks]

You should add the ability to start and stop. This means blending between the running cycle and the rest pose over a suitable number of frames. This should be implemented by starting the character in the rest pose, then using the space bar to start running, and using it again to stop the character.

3 Physical Simulation [25 marks]

In this part of the assignment, you will implement the bouncing dodecahedra of doom. Implementing bouncing balls on a flat plane is fairly easy, as is collision detection with the character. Adding an uneven surface is somewhat trickier, while doing so with a non-spherical primitive such as a dodecahedron is difficult.
You should start off with a single ball dropped or launched from a known location, then later add some randomisation of where multiple balls drop.

3.1 Task IIa: Bouncing Ball [6 marks]

You have been provided with a spheroid model (it's actually a subdivided icosahedron, but it's close enough to a sphere for our purposes). Start off by dropping it from a height of 10m above the ground, i.e. at $0, 0, 10$). Thereafter, apply gravity and impulse forces, and use a coefficient of elasticity of 0.6.
You should see the ball bouncing upwards repeatedly. Note that a sphere will always contact the ground at a single point at the lowest point of the sphere, which greatly simplifies the problem. Also, since the ball is spherical, you do not need to worry about implementing rotations. Again, you will find Terrain::getHeight() useful.
For the collision test with the ground, it is acceptable to test whether the z-component of the centre of the sphere is above the terrain by less than its radius.

3.2 Task IIb: Collision Detection [4 marks]

Now extend your collision testing to test whether the ball hits the character (which may be moving). For simplicity, treat the character as a sphere 1.8m tall and 0.3m in radius. On collision, increment a counter, and print out a message.

3.3 Task IIc: Uneven Surfaces [5 marks]

The scene has been provided with three different surfaces for you to test with. Use the 'l' key to switch between them. Modify your programme to bounce with respect to the local normal of the surface.

For collision testing, note that the radius of the sphere is 1m, while the terrain is in 3m squares. It is therefore acceptable to test the sphere only against the ground directly under the sphere's centre of gravity.

3.4 Task IId: Bouncing Dodecahedra [10 marks]

Substitute a dodecahedron for the spherical ball, and get arbitrary bounces working. You will need to implement both impulse forces and rotation. It is easiest to get the linear impulse forces working first, then add the rotational impulse afterwards. Again, you can simplify the collision test by comparing only against the ground directly beneath the centre of the dodecahedron, but you will need to detect which vertex collides with the ground first.
No marks for this task will be given unless all of the other tasks are substantially complete.