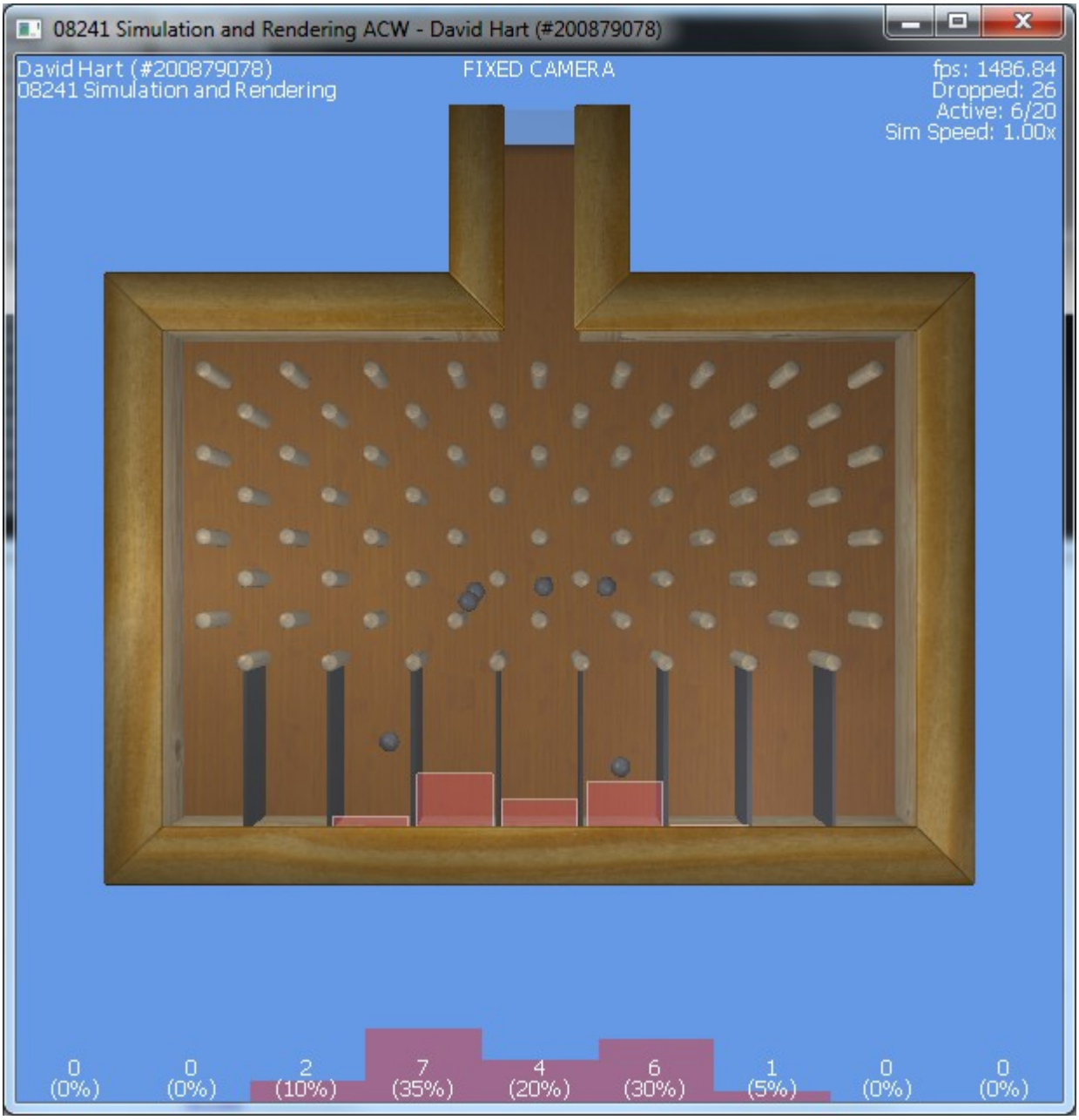
08241 Simulation and Rendering ACW



Usage instructions

- Pressing Key '1' spawns a ball at a random location. Key '2' spawns a ball every frame that the key is held down (20 balls may be active at any one time, this restriction is in place to prevent a massive performance hit with many balls).
- Pressing Key 'R' will reset the simulation, it will remove all active balls and reset statistics.
- Keys F1, F2 and F3 switch camera modes (as indicated above the viewport):

    - F1 Enables 'Fixed camera' mode, this mode has a fixed view over the machine
    - F2 switches to 'User camera' mode, in this mode the keys W, A, S and D control the movement of the camera, forwards, left, backwards and right, respectively. Holding down the right mouse button in this mode and dragging across the viewport will rotate the camera. Additionally, holding down the left mouse button in this mode will rotate the entire simulation.
    - F3 switches to 'Tracking camera' mode, in this mode the camera follows the last spawned ball, until it reaches the bottom.
- Key F5 and F6 disable and enable lighting, respectively.
- Key F7 and F8 enable and disable wireframe, repsectively.
- Numpad key + and – increase and decrease the simulation speed in increments of 0.1x the original speed. Sim speed is capped between 0-1x the original speed.

Features Implemented

For the rendering aspect of the coursework, I have implemented a .obj format model loader, which supports materials, loaded through the .mtl format and multiple meshes within a single .obj model. The model loader supports texture coordinates and vertex normals, so that the system can be expanded to use vertex buffer objects.

Additionally, I have implemented a texture loader which utilises the Gdiplus library, which is a standard library of Microsoft Windows. My implementation of texture loading uses Gdiplus to parse the images, so the pixel data can be obtained and transferred to an openGL texture.

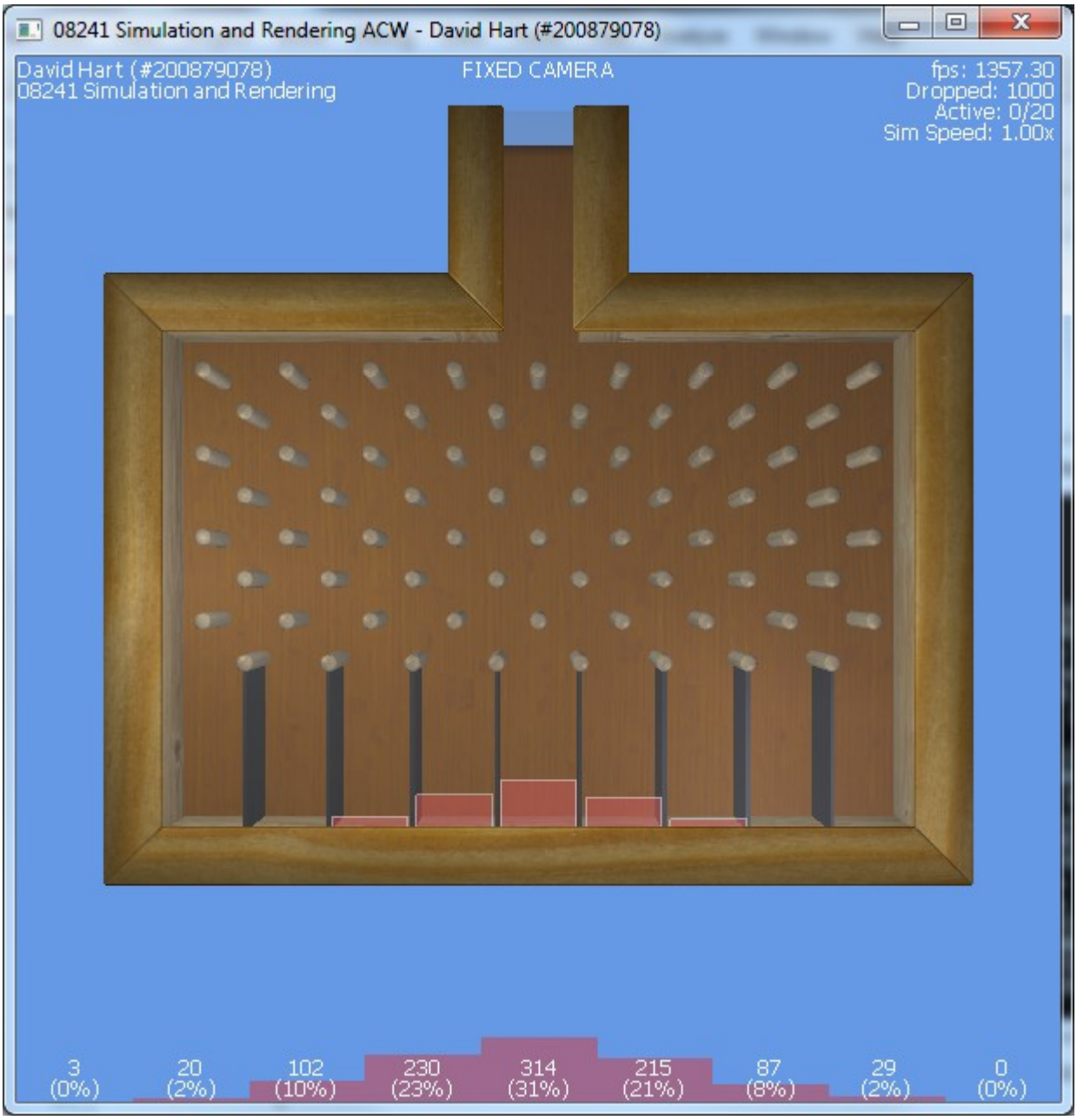
In order to display fonts, I have implemented a font loader and renderer. To load a font, I again use Gdiplus to render each glyph in a font (within a range) to a batch of Gdi images. I keep a note of the location of each glyph, as the information is required for rendering. I transfer each of the Gdi images that make up the font map, to an openGL texture. When rendering text, I loop through each letter of the font, drawing each glyph as a textured quad, using the texture coordinates I recorded when constructing the font map.

In order to speed up the rendering of these fonts, I implemented a class I have named SpriteBatch. It performs the job of grouping textured quad draw calls which use the same texture, into a single pair of glBegin and glEnd calls. This improved the performance of my font renderer, as multiple strings using the same font can be dispatched in a single draw call.

For the simulation aspect of the coursework, I have implemented collisions between balls and infinite planes, plane segments and other balls. This allows me to simulate a system with multiple balls colliding simultaneously. My collision detection methods, each determine if a collision will happen at the next time-step, and in such cases, further subdivides the time-step to try and determine approximately when a collision happens. When subdividing the time-step, I split the time slice in half each time, and determine if there is a collision at the midpoint, which tells me which half of the time-step the collision occurs in. I repeat this subdivision a number of times to increase the accuracy of my approximation.

I have implemented counters for each bucket the ball lands in, and a bar-graph display in

both 2D and 3D, to visualise the distribution of the balls. As can be seen in the screenshot below, dropping 1000 balls is enough to show that a bell curve is forming.



Further improvements

There are a number of ways in which my project can be further improved. More exact collision detection methods would give a much more accurate and reliable simulation. I could extend the collision detection and response code to allow for different types of balls, by adding variations to mass, elasticity and size.

I believe the collision detection and response code could be made much more efficient as currently each ball must be checked against 92 obstacles and every other ball in the simulation. If these checks could be cut down I could simulate more balls concurrently.