Framebuffers

- In OpenGL, there are various framebuffers.
- A framebuffer is simply a rectangular array of values.
- So far, we have used the standard display buffer and depth buffer
- We will look at other uses of framebuffers (in theory).

Blending

- So far, all objects are opaque ($\alpha = 1$).
- In order to render transparent objects, we need to blend.
- Basic idea: if two objects occupying the same pixel location has colour (R_1, G_1, B_1, A_1) and (R_2, G_2, B_2, A_2), the final result should be

$$c_1 \cdot (R_1, G_1, B_1, A_1) + c_2 \cdot (R_2, G_2, B_2, A_2)$$

  for some constants c_1 and c_2.
- It is often convenient to talk about source and destination: source is the fragment being rendered, and destination is what has already been rendered at the location.

Blending

- In the formula,

$$c_S \cdot (R_S, G_S, B_S, A_S) + c_D \cdot (R_D, G_D, B_D, A_D)$$

  we can make different choices for c_S and c_D.

- One common choice is $C_S = A_S$ and $C_D = (1 - C_S)$.

- e.g. coloured glass (source) will show a blend of the glass colour and the objects behind the glass (destination)

- Many other choices of C_S and C_D possible depending on desired effect.

Blending

- In order for blending to work, depth testing must be disabled. Why?
- Also, switching between source and destination will change the rendering outcome.
- General order
  - Render all opaque objects with depth testing on
  - Sort all transparent objects from back to front
  - Render transparent objects with depth testing on, but set the depth buffer to read-only (`glDepthMask`)

Framebuffer Objects

- In OpenGL, framebuffer objects can be created.
- We can choose to render into our own framebuffer objects off-screen
- We can also read the value at particular coordinates in any framebuffer objects.
- The values stored in each pixel does not have to be a colour (e.g. depth buffer)
- There are a number of useful applications.

Multi-pass Rendering

- So far, we have only talked about rendering objects to the display framebuffer.
- In multi-pass rendering, a scene can be rendered multiple times (in different ways) to obtain the final scene.
- Each pass can use the information from previous passes.
- e.g. environment maps to model reflection.

Ambient Occlusion

- We have assumed that ambient light is always uniform at all fragments.
- But even ambient light can be blocked.
- First pass: render the scene and record only normal vectors and depth at each pixel (normal map and depth map)
- Second pass: look at normal vectors and depths around the pixel. The ambient lighting is reduced by how many pixels around have smaller depths (in direction of normal).

Buffer Ping-Ponging

- We have used double buffering for smooth animation.
- But what is difficult to read values from the display buffers
- With our own framebuffers, we can easily use a pair of framebuffers.
- We render to one framebuffer, using information from the other framebuffer.
- Then we switch the roles of the two framebuffers.
- Useful if the animation models some process that is "evolving"
- e.g. simulating how a drop of food colouring spread in a pool of water

Picking

- In many applications, we would like to be able to use the mouse to select objects on screen.
- The input would be the window coordinates (x, y) from the mouse click.
- How do we actually know which object is selected?
- Issues:
  - need to "invert" from window coordinates to object coordinates
  - need to ignore objects hidden by depth testing

Picking

- Solution: create a separate framebuffer to render the objects in a parallel scene
- Ignore all lighting and texture calculations
- Each object and/or surface will be rendered in a unique solid colour.
- Simply read the colour at the picked location in this framebuffer.
- The unique colour tells us which object is picked.

Shadow Maps

- Previously, we have not consider the possibility of shadows.
- During lighting calculations, it is always assumed that there are no obstacles between the light source and the fragment being rendered.
- This is not realistic.

Shadow Maps

- How do we which object and/or fragment can receive light from a light source?
- We put a camera at the light source and render the scene!
- What the camera can see are the fragments that receives light.
- Record the distance/depth of those pixels that can be seen relative to the light source. This is the shadow map.

Shadow Maps

- Perform the actual rendering in a second pass.
- For each fragment, compare the distance between the fragment and the light against what is stored in the shadow map.
- If it is greater, then the light is blocked.

Shadow Maps

Some details:

- use perspective projection for point light source
- use orthographic projection for parallel light source
- the depth comparison needs to be done in "light camera" coordinate space: so we need model-view-projection matrices in that space

Anti-aliasing

- The computer screen is a rectangular array of pixels.
- Drawing a simple diagonal line can result in jaggedness if we simply turn a pixel on or off.
- This is particularly true at low resolution.
- If drawing a triangle, one way to smooth the shape boundaries is to shade based on how much of the pixel is contained in the shape.

Anti-aliasing

- One method: first render the scene into an internal framebuffer
- Then use multisampling:
  - for each pixel, sample a number of different values from the internal framebuffers around that location
  - the samples are combined (e.g. average)