CSE 167x Computer Graphics
Ramamoorthi UC San DiegoX Unit 2

1 OpenGL: Introduction

- OpenGL is a graphics API. It's like a high-level language for 3D CG that allows people to program their graphics hardware in a standardized way. (It serves as a layer of abstraction on top of this hardware.)
- The OpenGL rendering pipeline:
  1a. Vertices (e.g. for triangles) → a vertex shader, where primitive ops are performed on the geometry.
  1b. The input can also be images, and pixel operations can be performed on these images.
    - Images can be stored in texture memory, and applied to geometry in the scene.
    - Images can also be sent to the next stage in the same way as 1a's processed geometry.
  2. We then have to associate the shapes/vertices with pixels (*scan conversion/rasterization*).
    - The rasterizer is essentially the only thing specified by OpenGL (the rest is user-determined).
  3. After rasterizing (determining affected pixels), a fragment shader performs fragment operations.
  4. Finally, the output is written to the screen, which here is known as a framebuffer.
- This pipeline is now programmable (ever since $\sim$ 2003), i.e. as shaders.
  - These manifest as arbitrary programs that are executed on all vertices and/or all pixels.
  - GPU performance greatly outstrips CPU performance, even for non-graphics applications.
    * This has led to the notion of the GPGPU (general-purpose GPU), where the massive parallelization capability of a GPU is harnessed for arbitrary problems.
    * (The strength of a GPU is that it operates in parallel on all vertices or fragments.)
- Many buffers:
  - the front buffer, which we see
  - the back buffer (might write a complete drawing into this before swapping into the front buffer)
  - left and right buffers used for stereo and VR
  - the depth (z) buffer, used to determine which objects are in front of other objects
  - ...

  Drawing in OpenGL means *writing to a buffer* (in the most basic sense, the front and depth buffers).
- *Double buffering*: write a complete drawing to the back buffer, then swap with the front buffer.
- Buffer data/arrays are also used to store information about vertices.
- For portability reasons, OpenGL has no native windowing functionality.
  - Instead, we can use GLUT, Tcl/Tk, etc.
  - These toolkits utilize callbacks for interactions (with the mouse, keyboard, etc.).

2 OpenGL: Viewing

- There are two parts to viewing:
  - *Positioning the objects correctly in the scene*, with the **model view** transformation matrix
    - Model transformation: translation, rotation, etc. of the model itself
    - View transformation: transformation into the frame of the camera
  - *Projecting the 3D scene onto a 2D image*, with the **projection** transformation

3 OpenGL: Drawing

- Modern OpenGL has a reduced set of primitives: points (specified in homogeneous coordinates), lines (multiple can be chained together in a strip or loop), triangles, triangle strips (take one triangle, then add additional triangles by specifying one vertex at a time), and triangle fans (where triangles form a fan around one point).
  - `GL_POINTS`, `GL_LINES`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`
- In new OpenGL, we need one vertex array object (VAO) per object.

4 OpenGL: Shading

- Modern OpenGL includes tessellation shaders (which tessellate spline patches) and geometry shaders (which can add or remove new geometry to/from the scene) in addition to vertex and fragment shaders.
- The simplified OpenGL pipeline: user specifies vertices via vertex arrays, which are processed individually in parallel by the vertex shader (this also applies the model view and projection matrices); OpenGL then rasterizes each primitive (determines which pixels on the screen it corresponds to, and generates a *fragment* for each pixel that the primitive covers); the fragments are finally processed individually in parallel by the fragment shader.
  - The fragment shader typically involves shading and lighting calculations.
  - At the final stage, OpenGL will also manage the z-buffer depth test unless otherwise specified.
- Shader setup: one must be able to (1) create the shader (both vertex and fragment), (2) compile the shader, (3) attach the shader to the program, (4) link the program, and (5) use the program.
- Shader source is just a bunch of strings, which are compiled at runtime to be run on a GPU.
  - These strings can be read from a file.
- The output values of a vertex shader are interpolated (by the rasterizer) to fragments and sent to the fragment shader.
- Shading allows us to perceive 3D shape ("shape from shading").
  - Thus accurate lighting and shading are important to convey shape.
  - Note: the manner of shading is important too, e.g. *flat shading*, where each face takes a single color from a single vertex (and normals are the same across the face), gives a more faceted, polygonal (worse) appearance than *Gouraud/smooth shading*, where color and normals are interpolated across all of the vertices.
- Color:
  - RGB $\rightarrow$ primary colors, vertices of a color cube which can be combined in many ways

  - For example, R + G = yellow, while R + G + B = white
  - RGBA mode (8 bits per channel, 32 bits/4 bytes in total for each pixel) is often used in OpenGL
- *Vertex versus fragment shaders*: both can be used for lighting. Again, the vertex shader operates on vertices and the colors from that are interpolated across the pixels/fragments. The fragment shader receives inputs from the vertex shader (e.g. normals) and might do a lighting computation at each pixel/fragment. (If using a fragment shader, processing will happen at every pixel.)
  - Traditionally, lighting was handled on a per-vertex basis (cheaper if fewer vertices than fragments).
  - In today's hardware, fragment shaders are cheaper to write and allow for more interesting lighting calculations (e.g. highlights). Often, vertex count is also proportional to fragment count.
  - For instance, we can determine the normals at vertices, which will be interpolated across pixels, and then we can color each pixel according to a shading formula using the fragment shader.
    * This is called *Phong shading* (where we shade each pixel according to the interpolated normal at that pixel; note that the normal will be in 3D since it originated from vertices in 3D).
    * During Phong shading we can apply any illumination formula as part of the fragment shader, Torrance-Sparrow for example. It doesn't have to be Phong illumination, though it can be.

5 Lighting and Shading

- Gouraud shading involves bilinearly interpolating vertex values. However, there are errors associated with it, e.g. it can fail if the connection between vertices is curved; it is also not rotationally invariant.
  - It is therefore useful only for smooth (e.g. diffuse) shading effects.
- Phong shading is used for specular/glossy materials, i.e. materials with highlights based on lighting.
  - Note: *roughness* controls the width of the highlights and the perceived shininess of the object; greater roughness means wider highlights and a less shiny-looking object.
  - To create highlights, use Phong shading with Phong illumination. In other words, *interpolate the vertex normals/positions instead of the vertex colors.* Then do lighting computations (apply the Phong illumination formula) in the fragment shader based on the normals and positions.
- The difference between Gouraud and Phong shading: Gouraud shading is performed entirely in the vertex shader, meaning it's dependent on geometry, while Phong shading also uses the fragment shader.
- Shading depends on (1) illumination in the scene and (2) object material properties.
- Types of light sources:
  - *Point light source*:
    * a point, so has a position in space (3D vector) and a color (RGBA)
    * comes with an attenuation (quadratic) model:

$$\frac{1}{k_c + k_l d + k_q d^2}$$

      d is the distance to the light source; the k's are constants.

      Light energy falls off according to the inverse-square law, i.e. a point at distance d receives intensity proportional to $1/d^2$. (To think about this, imagine an equal amount of energy is distributed evenly over the surface of a radius-d sphere. A sphere's surface area is $4\pi d^2$, so as d increases there's quadratically less energy being given to any one point – since it's split over quadratically more points.) **This is where the $k_q d^2$ term comes from.**

      If the light is a line light source (light coming off of a tube or something), light falls off linearly (as $1/d$). **This is where the $k_l d$ term comes from.**

Finally, if everything is close to an area light source, we might want constant attenuation. **This is where the k_c term comes from** (as an approximation).

k_q controls quadratic attenuation, k_l controls linear attenuation, and k_c controls constant attenuation. If we just had a point light source, we might want $k_q = 1$ and the others zero; if we just had a line light source, we might want $k_l = 1$ and the others zero; if we had a distant, directional light source, we might want $k_c = 1$ and the others zero.

Often we assume constant attenuation ($k_c = 1$) despite it being physically incorrect.

- *Directional light source*:
  - source at infinity ("distant relative to the scene") pointing in a direction, e.g. sun
  - represent within point source framework by setting homogeneous coordinate to 0
  - **no attenuation**; set $k_c = 1$ and the others equal to zero

- Material properties:
  - How does a surface reflect light? For this we need to know the surface normals.
  - Usually specify the normal at each vertex and have the rasterizer interpolate it to each fragment.
    - For parametric surfaces, we can compute normals manually.
    - For complex objects, we can average the face normals to obtain vertex normals.
  - There are four basic lighting terms:
    - *Ambient*: light that's always there.
      - "There's always light bouncing around the room, giving it a general diffused feeling."
      - A hack for global illumination (which would simulate the exact distribution of light).
      - Light uniform, coming from all directions → **represented as global constant**.

      (RGBA `vec4` ambient color)

    - *Diffuse*: light reflecting equally in all directions (rough, matte Lambertian surfaces).
      - Has cosine falloff: if θ is the angle between the incoming light ray $\mathbf{l}$ and the normal $\mathbf{n}$, then output light has intensity proportional to $\mathbf{l} \cdot \mathbf{n} = \max(\cos\theta, 0)$. The max is because light cannot be received from below the surface (from the side opposite the normal); i.e. θ cannot be between 90 and 270 degrees.

      (RGBA `vec4` surface diffuse color) $*$ (RGBA `vec4` light color) $*$ $(\max(\mathbf{n} \cdot \mathbf{l}, 0))$

    - *Specular*: light appearing according to both the incident and viewing direction.
      - Light reflects close to the mirror direction (angle of reflection equal to angle of incidence).
      - The difference between the eye and mirror angles affects the amount of light observed.
      - *Phong illumination*: represent view-dependent highlights using the dot product (cosine) between the vector to the eye and the reflection of the lighting direction about the normal. Also raise the cosine to some power to control shininess/roughness.
      - *Blinn-Phong model*: take dot product of *normal* and *half vector between lighting direction and eye direction*, and can again raise it to some power for control.
      - When shininess is 1 (just a cosine), it's similar to diffuse lighting (the specular highlight is large and diffuse). As shininess becomes greater (cosine raised to a power > 1), the specular highlight becomes smaller and more intense. And when shininess is infinite, the specular highlight is nonzero only when viewed from the mirror direction.

      (RGBA `vec4` surface specular) $*$ (RGBA `vec4` light) $*$ $(\max(\mathbf{n} \cdot \text{halfvec}, 0))^{\text{shininess}}$

        * *Emissive*: light that you get from looking *directly* at a light source.
            · In OpenGL, must create geometry for the light source in order to see it.
            · Geometry is lit by light sources, but the light source itself is not seen.
- Lights transform in the same way as other geometry; the difference is that lights are in 3D, so only the model view matrix is applied. (This is one of the few times we care about separating MV and P.)

6 OpenGL: Matrix Stacks

- OpenGL vertex transforms:
  - Each (x, y, z, w) vertex in object coordinates is transformed through
    * *the model view matrix* (object transforms & transformation into the camera frame), at which point we have the eye coordinates used for lighting,
    * *the projection matrix* (3D to 2D), at which point we have clip coordinates (anything outside of the clipping volume will be clipped; the clipping volume is the $[-1, 1]$ cuboid in NDC),
    * *perspective division (de-homogenization)*, at which point we have NDC, and
    * *a viewport transform*, at which point we have window coordinates.
- The model view and projection transforms are usually represented using matrix stacks.
- Matrix stacks are useful for hierarchically defined objects where every transform can be defined relative to a parent transform, or when we have different transforms for multiple instances of the same geometry.
- The last transform added to the stack should be the first applied (right matrix multiplication).

7 OpenGL: Z-Buffering

- OpenGL uses a z-buffer for depth testing. We maintain an additional buffer for the depth (for each pixel, the nearest z value to the camera). So each pixel now has a depth in addition to a color. If multiple fragments map to the same region, only the closer one (as its depth and color) is maintained.

8 OpenGL: Texture Mapping

- In texture mapping, we use images (instead of more polygons with color) to get fine-scale texture detail.
- Each vertex should have a texture coordinate associated with it; then, during rasterization, the texture coordinates will be interpolated at each fragment. Then texture can be added in the fragment shader.
- Mipmaps are texture maps at multiple resolutions (fine to coarse).