3D Graphics Basics

Overall Idea

1. Define (using math) a scene of virtual objects in a 3D coordinate system.
2. Generate a 2D picture which simulates a picture taking of the scene from some camera position in the coordinate system. That is, calculate relevant 2D data from the 3D data of step 1. This step is often called rendering the scene.

Overall Idea

1. Define (using math) a scene of virtual objects in a 3D coordinate system.
2. Generate a 2D picture which simulates a picture taking of the scene from some camera position in the coordinate system. That is, calculate relevant 2D data from the 3D data of step 1. This step is often called rendering the scene.

Note that both steps are *mathematical*. Thus, 3D graphics is mathematical at its core.

In principle, we can do 3D graphics without a computer. However, for efficiency reasons, we normally involve a computer to help us do the calculations.

Overall Idea

1. Define (using math) a scene of virtual objects in a 3D coordinate system.
2. Generate a 2D picture which simulates a picture taking of the scene from some camera position in the coordinate system. That is, calculate relevant 2D data from the 3D data of step 1. This step is often called rendering the scene.

Note that both steps are *mathematical*. Thus, 3D graphics is mathematical at its core.

In principle, we can do 3D graphics without a computer. However, for efficiency reasons, we normally involve a computer to help us do the calculations.

In this course, we will study the principles (the math and the algorithms) of the most widespread way of performing the two steps above. The exam will be focusing on this part.

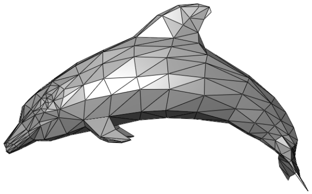
We will also get some hands-on experience on how to program and execute these principles with current technology.

Step 1: Defining 3D Objects

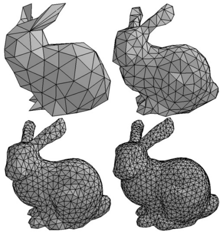
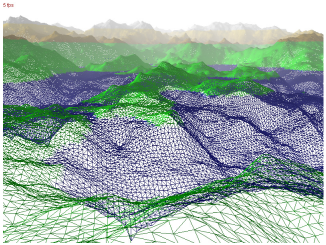
In real life, we mostly meet opaque objects, for which we only see their surfaces $\Rightarrow$ we focus on defining surfaces in 3D.

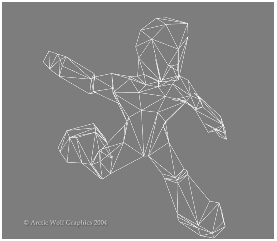
A very simple way to define a surface in 3D is via three 3D points. They define a plane and a triangle in that plane.

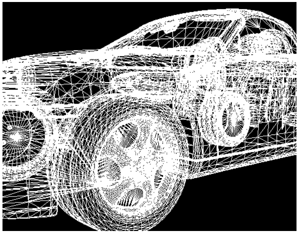
In our story, triangles will be a fundamental element. From these, we can build larger surfaces:



More Examples

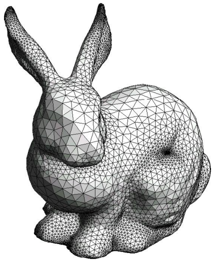







3D Models

3D model = list of 3D points and info on how they group into triangles.

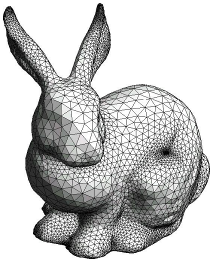


$$
\begin{aligned}
x_1 &= (10, 10, 10) \\
x_2 &= (10, 20, 30) \\
&\vdots \\
T_1 &= (x_1, x_2, x_3) \\
T_2 &= (x_1, x_2, x_4) \\
&\vdots
\end{aligned}
$$

Besides 3D position, more information such as color values, normal vectors, and texture coordinates, may be included in vertices for use during rendering.

3D Models

3D model = list of 3D points and info on how they group into triangles.



$$\begin{aligned} x_1 &= (10, 10, 10) \\ x_2 &= (10, 20, 30) \\ &\vdots \\ T_1 &= (x_1, x_2, x_3) \\ T_2 &= (x_1, x_2, x_4) \\ &\vdots \end{aligned}$$

Besides 3D position, more information such as color values, normal vectors, and texture coordinates, may be included in vertices for use during rendering.

The generation of 3D models is an artistic endeavour (using software tools such as Blender, Maya, etc.) and is outside of computer science.

In our course, we assume 3D models are provided by others (or we use simple, geometric ones like boxes).

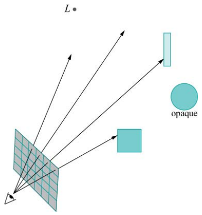
Step 2: Generate a 2D Picture

The goal is to "paint the view" of the scene from the given camera position.

A picture is a 2D array of pixels. We need to find a color for each pixel.

Plan:

- Identify the picture/screen with a part of a plane in the scene.
- Calculate a color value of a pixel based on scene contents along its ray—often just on the closest object in the scene along the ray (modeling an opaque object and clear air).



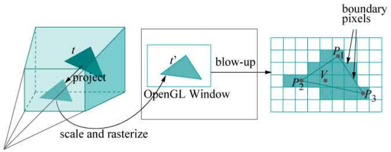
Step 2: Generate a 2D Picture

The most direct execution of this idea is *ray-driven:*

For each pixel/ray in picture: calculate a color for that pixel.

Another way is *triangle-driven:*

For each triangle in the scene: find the pixels it covers on the screen (rasterization) and calculate a color for each of those.



Both versions need a way to determine the closest object along each ray.

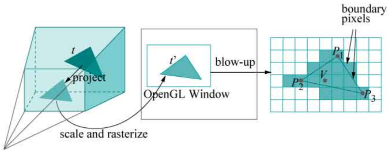
Step 2: Generate a 2D Picture

The most direct execution of this idea is *ray-driven:*

For each pixel/ray in picture: calculate a color for that pixel.

Another way is *triangle-driven:*

For each triangle in the scene: find the pixels it covers on the screen (rasterization) and calculate a color for each of those.



Both versions need a way to determine the closest object along each ray.

For efficiency reasons, the triangle-driven method is the most widespread, in particular for real-time applications (it can be implemented by simple and easy parallelizable computations). OpenGL (and this course) is based on this method.

The Graphics Pipeline

The Graphics Pipeline

- Compose the scene from the provided models. This requires *scaling*, *rotation*, and *translation* of the models.

The Graphics Pipeline

- Compose the scene from the provided models. This requires *scaling*, *rotation*, and *translation* of the models.
- Define a *camera position*.

The Graphics Pipeline

- Compose the scene from the provided models. This requires *scaling*, *rotation*, and *translation* of the models.
- Define a *camera position*.
- *Project* the points of the models of the scene onto a 2D plane (given by the camera position). The plane is identified with the screen.

The Graphics Pipeline

- Compose the scene from the provided models. This requires *scaling*, *rotation*, and *translation* of the models.
- Define a *camera position*.
- *Project* the points of the models of the scene onto a 2D plane (given by the camera position). The plane is identified with the screen.
- For each triangle perform *rasterization*: find the pixels it covers in the screen. *Interpolate* any extra data (color, etc.) in the three vertices to find data values for the pixels.

The Graphics Pipeline

- Compose the scene from the provided models. This requires *scaling*, *rotation*, and *translation* of the models.
- Define a *camera position*.
- *Project* the points of the models of the scene onto a 2D plane (given by the camera position). The plane is identified with the screen.
- For each triangle perform *rasterization*: find the pixels it covers in the screen. *Interpolate* any extra data (color, etc.) in the three vertices to find data values for the pixels.
- For each of these pixels: calculate a color (*shading*).

The Graphics Pipeline

- Compose the scene from the provided models. This requires *scaling*, *rotation*, and *translation* of the models.
- Define a *camera position*.
- *Project* the points of the models of the scene onto a 2D plane (given by the camera position). The plane is identified with the screen.
- For each triangle perform *rasterization*: find the pixels it covers in the screen. *Interpolate* any extra data (color, etc.) in the three vertices to find data values for the pixels.
- For each of these pixels: calculate a color (*shading*).
- Apply the color to the pixel on screen in a suitable way (e.g. if other triangles are closer, discard the color (using a *z-buffer*)).

The Graphics Pipeline

- Compose the scene from the provided models. This requires *scaling*, *rotation*, and *translation* of the models.
- Define a *camera position*.
- *Project* the points of the models of the scene onto a 2D plane (given by the camera position). The plane is identified with the screen.
- For each triangle perform *rasterization*: find the pixels it covers in the screen. *Interpolate* any extra data (color, etc.) in the three vertices to find data values for the pixels.
- For each of these pixels: calculate a color (*shading*).
- Apply the color to the pixel on screen in a suitable way (e.g. if other triangles are closer, discard the color (using a *z-buffer*)).

The Graphics Pipeline

- Compose the scene from the provided models. This requires *scaling*, *rotation*, and *translation* of the models.
- Define a *camera position*.
- *Project* the points of the models of the scene onto a 2D plane (given by the camera position). The plane is identified with the screen.
- For each triangle perform *rasterization*: find the pixels it covers in the screen. *Interpolate* any extra data (color, etc.) in the three vertices to find data values for the pixels.
- For each of these pixels: calculate a color (*shading*).
- Apply the color to the pixel on screen in a suitable way (e.g. if other triangles are closer, discard the color (using a *z-buffer*)).

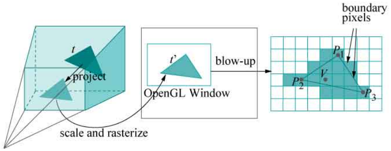
Important to note: Triangles are simply (triples of) vertices until the rasterization phase. In other words:

Until rasterization, our data are essentially points.

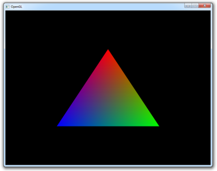
We will see the details of all the terms in italics (and many more) later.

Example of Interpolation

During rasterization,



for instance color values included with the three vertices may be interpolated for pixels across the triangle:



The same holds for other data coming with the vertices, such as normal vectors and texture coordinates. More details of interpolation later.

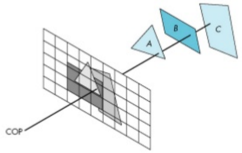
Hidden Surface Removal with the z-buffer

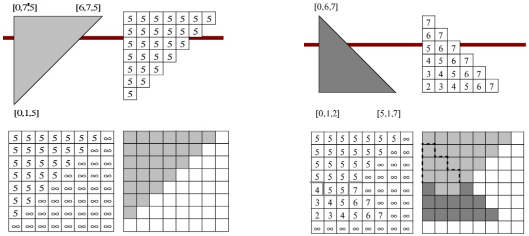
Colorbuffer = screen-size 2D buffer of color values
z-buffer = screen-size 2D buffer of depth values

Hidden Surface Removal with the z-buffer

Colorbuffer = screen-size 2D buffer of color values
z-buffer = screen-size 2D buffer of depth values

```
f = fragment
(i,j) = screen position of f
z = depth in scene of f
IF z < zbuffer[i,j]:
    colorbuffer[i,j] = f.color
    zbuffer[i,j] = z
```





5	5	5	5	5	5	5
5	5	5	5	5	5	
5	5	5	5	5		
5	5	5	5			
5	5	5				
5	5					
5						

5	5	5	5	5	5	5	∞
5	5	5	5	5	5	∞	∞
5	5	5	5	5	∞	∞	∞
5	5	5	5	∞	∞	∞	∞
5	5	5	∞	∞	∞	∞	∞
5	5	∞	∞	∞	∞	∞	∞
5	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞

7					
6	7				
5	6	7			
4	5	6	7		
3	4	5	6	7	
2	3	4	5	6	7

5	5	5	5	5	5	5	∞
5	5	5	5	5	5	∞	∞
5	5	5	5	5	∞	∞	∞
5	5	5	5	∞	∞	∞	∞
4	5	5	7	∞	∞	∞	∞
3	4	5	6	7	∞	∞	∞
2	3	4	5	6	7	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞

Some History

1960-70-80s	Many of the fundamental math ideas and algorithms of 3D graphics are developed. Executed on the CPU.
1990s	GPUs with hardware support for (some of) the 3D algorithms. Provides speed via parallelization.
2000s	GPUs become programmable (as opposed to supporting a fixed set of algorithms), are still built to support mainly 3D operations.
2010s	GPUs support more general-purpose calculations (not relevant for this course).

APIs are needed for programming the hardware. Examples include OpenGL, DirectX, Vulkan, Metal.

In contrast to many other areas of programming, APIs of 3D graphics are moving towards *less* abstraction (to exploit current CPUs the fullest).

In this course, we use the widespread, cross-platform API OpenGL (1992-), accessed in Java via the wrapper API JOGL.