Overview

- An algorithm for drawing picture
- What is *rasterization*? What is *ray tracing*?
- What is *fragment*? What is *buffer*? What is *shader*?
- How do you *describe a shape*?
- What is *GPU*?
The Main Algorithm
How to draw pictures algorithmically?
How to draw pictures algorithmically?

• Prescribe/input:

• Output:
Big question

How to draw pictures algorithmically?

• Prescribe/input:
  ▶ Geometries (triangle mesh) in the scene
  ▶ Color, material,… of each geometry
  ▶ Light
  ▶ Camera

• Output:
  ▶ Color per pixel in the screen
Main algorithm

• Prescribe/input:
  ▶ Geometries (triangle mesh) in the scene
  ▶ Color, material,… of each geometry
  ▶ Light
  ▶ Camera

Step 1
Determine which triangle corresponds to which pixel

Step 2
Color each pixel according to lighting/material/orientation etc.

• Output:
  ▶ Color per pixel in the screen
From 3D to 2D

3D scene

center of projection

screen

these are “camera”
From 3D to 2D

• Working out which geometries correspond to which pixels can be tedious
In week 3, we will learn that there is always a transformation that normalizes the frustum into a standard box.

The question remains. How do we turn triangles to pixels?
Scene-Screen Incidence Relation

- Determine which triangles of the scene is incident to which pixels of the screen

![Diagram showing the relationship between scene and screen elements.](image)
Scene-Screen Incidence Relation

• Determine which triangles of the scene is incident to which pixels of the screen

Definition

A **fragment** is a triangle-pixel pair so that the pixel is incident to (covered by) the triangle.

• Each fragment knows which triangle it comes from
• Each fragment knows which pixel to land on
We are constructing a mapping between scene and screen.

There are 2 main ways for traversing and listing out all the fragments.
**Scene-Screen Incidence Relation**

**Method 1**

```plaintext
for each triangle in scene
  for each pixel in screen
    output the pair if the triangle occupies that pixel.
  end for
end for
```

**Method 2**

```plaintext
for each pixel in screen
  for each triangle in scene
    output the pair if the triangle occupies that pixel.
  end for
end for
```
Scene-Screen Incidence Relation

Method 1
for each triangle in scene
for each pixel in screen
output the pair if the triangle occupies that pixel.
end for
end for

Method 2
for each pixel in screen
for each triangle in scene
output the pair if the triangle occupies that pixel.
end for
end for

We will learn this math problem in Week5:

What is the criterion that the pixel center is in the triangle?
Scene-Screen Incidence Relation

**Method 1**
for each triangle in scene
   for each pixel in screen
      output the pair if the triangle occupies that pixel.
   end for
end for

**Method 2**
for each pixel in screen
   for each triangle in scene
      output the pair if the triangle occupies that pixel.
   end for
end for
Scene-Screen Incidence Relation

**Rasterization**

for each triangle in scene
  for each pixel in screen
    output the pair if the
    triangle occupies that
    pixel.
  end for
end for

**Ray casting/tracing**

for each pixel in screen
  for each triangle in scene
    output the pair if the
    triangle occupies that
    pixel.
  end for
end for

Rasterization v.s. Ray tracing

**Rasterization**
- Easy to speed up
- Hardcoded in hardware
- Real-time rendering
- No photorealism

**Ray casting/tracing**
- Hard to speed up  
  (need nontrivial data structure)
- Hardware available recently
- Recursive ray tracing => photorealism
Rasterization v.s. Ray tracing

**Rasterization**
- Easy to speed up
- Hardcoded in hardware
- Real-time rendering
- No photorealism

**Ray casting/tracing**
- Hard to speed up (need nontrivial data structure)
- Hardware available recently
- Recursive ray tracing => photorealism
Rasterization v.s. Ray tracing

**Rasterization**
- Easy to speed up
- Hardcoded in hardware
- Real-time rendering
- No photorealism

**Ray casting/tracing**
- Hard to speed up (need nontrivial data structure)
- Hardware available recently
- Recursive ray tracing $\Rightarrow$ photorealism

First few weeks
- Runs on GPU
- OpenGL API
Back to the main algorithm
Raster graphics pipeline

for each triangle in scene
for each pixel in screen
output the pair if the triangle occupies that pixel.
end for
end for
Raster graphics pipeline

Pseudocode

```
for each triangle in scene
  for each pixel in screen
    output the pair if the triangle occupies that pixel.
  end for
end for
```

Data in allocated memory

a.k.a. buffer
Raster graphics pipeline

**rasterizer**

```plaintext
for each triangle in scene
  for each pixel in screen
    output the pair if the triangle occupies that pixel.
  end for
end for
```
for each triangle in scene
for each pixel in screen
output the pair if the triangle occupies that pixel.
end for
end for

Raster graphics pipeline

rasterizer
Raster graphics pipeline

**rasterizer**

```
for each triangle in scene
  for each pixel in screen
    output the pair if the triangle occupies that pixel.
  end for
end for
```
Raster graphics pipeline

for each triangle in scene
  for each pixel in screen
    output the pair if the triangle occupies that pixel.
  end for
end for

for each fragment
  ... compute color
  ... assign frag_color
end for
Raster graphics pipeline

**rasterizer**

\[
\text{for each triangle in scene}
\text{for each pixel in screen}
\text{output the pair if the triangle occupies that pixel.}
\text{end for}
\text{end for}
\]

**fragment shader**

\[
\text{for each fragment}
\text{... compute color}
\text{... assign \text{frag\_color}}
\text{end for}
\]

**resolve depth**

\[
\text{for each pixel}
\text{keep only the top fragment}
\text{end for}
\]

**list of fragments**

![List of fragments](image)

**list of fragments**

![List of fragments](image)

**color per pixel**

![Color per pixel](image)
Raster graphics pipeline

For each triangle in scene
For each pixel in screen
If the triangle occupies that pixel
Output the pair
End for
End for

Rasterizer

List of fragments

For each fragment
... compute color
... Assign frag_color
End for

Fragment shader

List of fragments

Resolve depth

For each pixel
... keep only the top fragment
End for

Color per pixel

Show on screen or Save as image

Frame buffer
Raster graphics pipeline

For each triangle in scene
For each pixel in screen
Output the pair if the triangle occupies that pixel.
End for
End for

How do we organize the data that describes a triangle mesh?
Description of triangle mesh

- Every shape (e.g. square) is partitioned into triangles mesh
- Triangle mesh consists of
  - Vertices
  - Connectivity
Description of triangle mesh

- Every shape (e.g. square) is partitioned into triangles mesh

- Triangle mesh consists of
  - Vertices
    - Position
    - Other attributes like color/material
  - Connectivity
    - Pointers to 3 vertices per triangle
Description of triangle mesh

• Every shape (e.g. square) is partitioned into triangles mesh

• Triangle mesh consists of
  ▶ Vertices
  - Position
  - Other attributes like color/material
  we call the location storing these data the **vertex buffer**
  ▶ Connectivity
  - Pointers to 3 vertices per triangle
  we call the location storing these data the **index buffer**
Description of triangle mesh

\[ (-\frac{1}{2}, \frac{1}{2}) \quad (\frac{1}{2}, \frac{1}{2}) \]

\[ (-\frac{1}{2}, -\frac{1}{2}) \quad (\frac{1}{2}, -\frac{1}{2}) \]
### Description of triangle mesh

- Give an arbitrary order for vertices.
- Give an arbitrary order for triangles.

<table>
<thead>
<tr>
<th>Vertex buffer</th>
<th>Index buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0  -0.5, -0.5,</td>
<td>0  0, 1, 3,</td>
</tr>
<tr>
<td>1  0.5, -0.5,</td>
<td>1  2, 3, 1</td>
</tr>
<tr>
<td>2  0.5,  0.5,</td>
<td></td>
</tr>
<tr>
<td>3  -0.5,  0.5</td>
<td></td>
</tr>
</tbody>
</table>

Here is a diagram illustrating the vertex and index buffers:

- Vertex buffer:
  - pt0: \((-0.5, -0.5, 0.5, 0.5, -0.5, 0.5)\)
  - pt1: \((0.5, -0.5, 0.5, 0.5, -0.5, 0.5)\)
- Index buffer:
  - Triangle 0: 0, 1, 3
  - Triangle 1: 2, 3, 1

The coordinates of the vertices and triangles are shown in the diagram.
Description of triangle mesh

Every geometry is completely described in a *geometry spreadsheet* (and an instruction how to parse the vertex buffer)

**Geometry spreadsheet**
(a.k.a. *vertex array object (VAO)*)

<table>
<thead>
<tr>
<th>Vertex buffer(s)</th>
<th>Index buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute</td>
<td>vertex</td>
</tr>
<tr>
<td>vertex</td>
<td>triangle</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>-0.5 -0.5 0.0 1.0</td>
<td>0 1 3</td>
</tr>
<tr>
<td>0.5 -0.5 0.0 1.0</td>
<td>2 3 1</td>
</tr>
<tr>
<td>0.5 0.5 0.0 1.0</td>
<td>3 4 6</td>
</tr>
<tr>
<td>-0.5 0.5 0.0 1.0</td>
<td>5 6 4</td>
</tr>
<tr>
<td>0.0 0.0 1.0 1.0</td>
<td>10 12 7</td>
</tr>
<tr>
<td>0.0 0.0 -1.0 1.0</td>
<td>8 9 11</td>
</tr>
<tr>
<td>-0.5 -0.5 0.0 1.0</td>
<td>10 11 13</td>
</tr>
<tr>
<td>0.5 -0.5 0.0 1.0</td>
<td>12 13 11</td>
</tr>
<tr>
<td>0.5 0.5 0.0 1.0</td>
<td>13 14 16</td>
</tr>
<tr>
<td>-0.5 0.5 0.0 1.0</td>
<td>15 16 14</td>
</tr>
<tr>
<td>0.0 0.0 1.0 1.0</td>
<td>20 22 17</td>
</tr>
<tr>
<td>0.0 0.0 -1.0 1.0</td>
<td>18 19 21</td>
</tr>
<tr>
<td>2.0 0.0</td>
<td></td>
</tr>
<tr>
<td>1.0 1.0</td>
<td></td>
</tr>
<tr>
<td>1.5 -1.5</td>
<td></td>
</tr>
<tr>
<td>1.0 1.0</td>
<td></td>
</tr>
<tr>
<td>0.5 0.5</td>
<td></td>
</tr>
<tr>
<td>2.0 0.0</td>
<td></td>
</tr>
<tr>
<td>1.0 1.0</td>
<td></td>
</tr>
<tr>
<td>1.5 -1.5</td>
<td></td>
</tr>
<tr>
<td>1.0 1.0</td>
<td></td>
</tr>
<tr>
<td>0.5 0.5</td>
<td></td>
</tr>
<tr>
<td>0.5 0.5</td>
<td></td>
</tr>
<tr>
<td>0.5 0.5</td>
<td></td>
</tr>
</tbody>
</table>
For each triangle in scene
For each pixel in screen
Output the pair if the triangle occupies that pixel.
End for
End for

Raster graphics pipeline
Raster graphics pipeline

for each triangle in scene
for each pixel in screen
output the pair if the triangle occupies that pixel.
end for
end for

vertex buffer

index buffer

rasterizer

fragment shader

list of fragments

for each fragment
... compute color...
assign frag_color
end for
Recall that before the rasterization, we need to perform some transformation…
Raster graphics pipeline

**vertex shader**

```
for each vertex
parse vert. attrib.
compute transform
... output coordinate in the viewing box
end for
```

**rasterizer**

```
for each triangle in scene
  for each pixel in screen
    output the pair if the triangle occupies that pixel.
  end for
end for
```

**vertex buffer**

| -0.5 | -0.5 | 0.0 | 1.0 | 2.0 | 0.0 |
| 0.5  | -0.5 | 0.0 | 1.0 | 1.0 | 1.0 |
| 0.5  | 0.5  | 0.0 | 1.0 | 1.0 | 1.0 |
| -0.5 | 0.5  | 0.0 | 1.0 | 1.0 | 1.0 |
| 0.0  | 0.0  | 1.0 | 1.0 | 0.5 | 0.5 |
| 0.0  | 0.0  | -1.0| 1.0 | 0.5 | 0.5 |
| -0.5 | -0.5 | 0.0 | 1.0 | 2.0 | 0.0 |
| 0.5  | -0.5 | 0.0 | 1.0 | 1.0 | 1.0 |
| 0.5  | 0.5  | 0.0 | 1.0 | 1.5 | -1.5 |
| -0.5 | 0.5  | 0.0 | 1.0 | 1.0 | 1.0 |

**index buffer**

| 0  | 1  | 3  |
| 2  | 3  | 1  |
| 3  | 4  | 6  |
| 5  | 6  | 4  |
| 10 | 12 | 7  |
| 8  | 9  | 11 |
| 10 | 11 | 13 |
| 12 | 13 | 11 |
| 13 | 14 | 16 |
| 15 | 16 | 14 |

**list of fragments**

for each fragment
... compute...
... assign...
end for
Raster graphics pipeline

**vertex shader**

```plaintext
for each vertex
    parse vert. attrib.
    compute transform
    ...
output coordinate in the viewing box
end for
```

**rasterizer**

```plaintext
for each triangle in scene
    for each pixel in screen
        output the pair if the triangle occupies that pixel.
        end for
    end for
```

**fragment shader**

```plaintext
for each fragment
    ...
    compute color
    ...
    assign frag_color
end for
```

**resolve depth**

```plaintext
for each pixel
    keep only the top fragment
end for
```

**color per pixel**
Raster graphics pipeline

vertex shader

for each vertex
parse vert. attrib.
compute transform
... 
output coordinate in the viewing box
end for

fixed, hardcoded

rasterizer

for each triangle in scene
for each pixel in screen
output the pair if the triangle occupies that pixel.
end for
end for

flexible

fragment shader

for each fragment
... 
compute color
... 
assign frag_color
end for
end for

color per pixel

for each pixel
keep only the top fragment
end for

flexible

• Flexible: Frequently reprogrammed depending on application
• Fixed: Hardcode in a chip
• Every for each loop can be parallelized
Graphics Processing Unit (GPU)
What is GPU?

- Hardware that accelerates the graphics pipeline.
- GPU is a cluster of thousands of efficient workers knowing simple arithmetics working in parallel.
- GPU works in “Single Instruction Multiple Data” (SIMD) (as opposed to Single Instruction Single Data (SISD) like CPU)
What is GPU?

A 19th century office building

(image courtesy Jean Feydy)
What is GPU?

A 19th century office building

(image courtesy Jean Feydy)
What is GPU?

(image from Real-Time Rendering Ch 23)
What is GPU?

- Frequent use of graphics pipeline → Dedicated hardware
- GPU manufacturers provide application programming interface (API) which is a list of functions allowing us to command GPU
- 90’s: GPU are made with fixed functions (e.g. Legacy OpenGL)
- Mid 2000’s: Programmable shaders (e.g. Modern OpenGL)

**Definition** A *shader* is a program that runs on GPU

- Trend: Towards programmability and flexibility
- General purpose GPU (e.g. CUDA), high performance computing, parallel computing, machine learning
Summary
- **Fragments** are triangle-pixel incidence/intersection
- **Rasterization** and **ray tracing** are two ways of looping to find fragments
- A **buffers** is an allocated memory (on the graphics card)
- A **shader** is a program running on GPU
- Describe a shape by a geometry spreadsheet (vertex array object) stored in **vertex buffer** and **index buffer**
- Raster graphics pipeline:

  - Implemented in GPU; **OpenGL** is an API to command it.