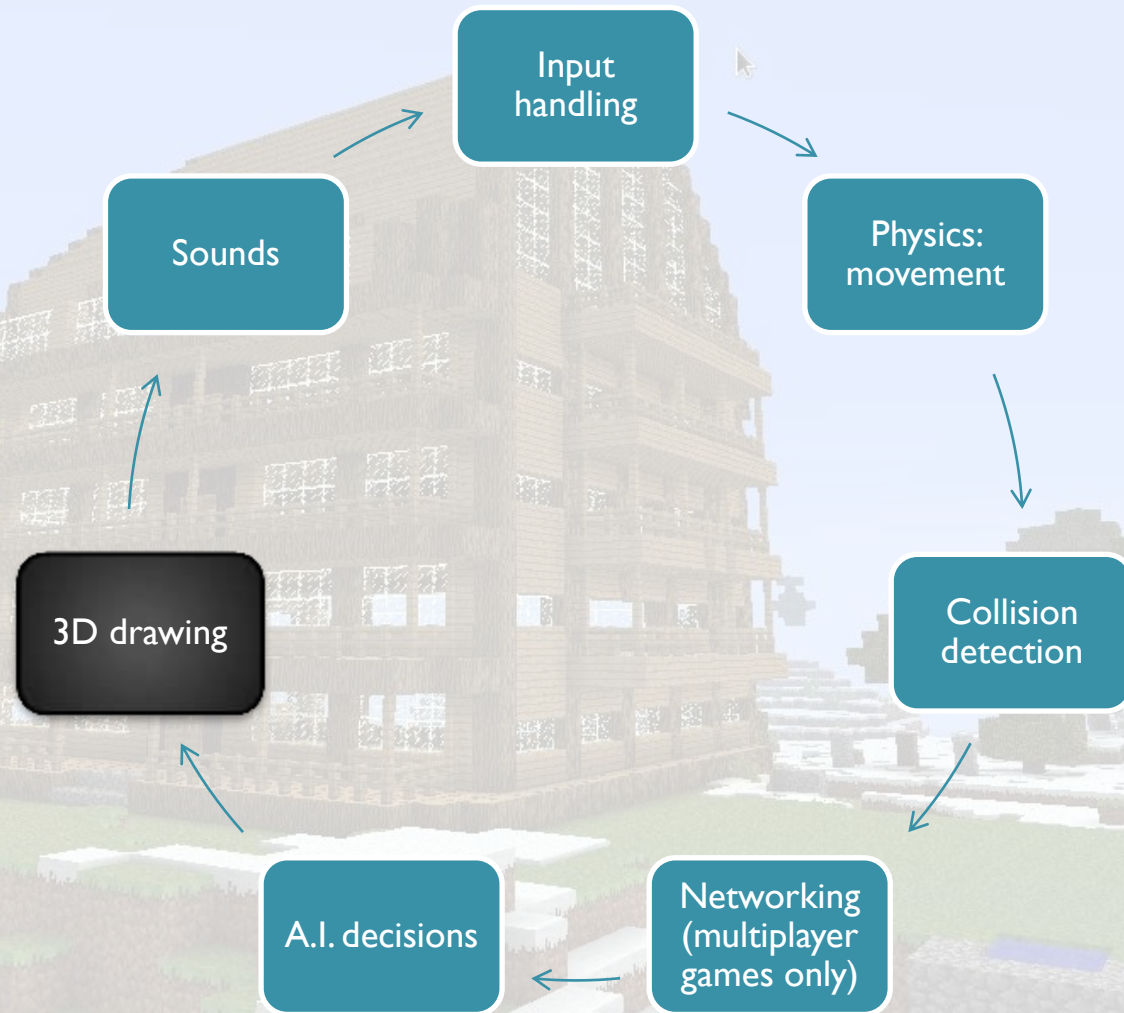


Overview of 3D Graphics Programming

Henry Wise Wood
Math Club
March 12, 2012

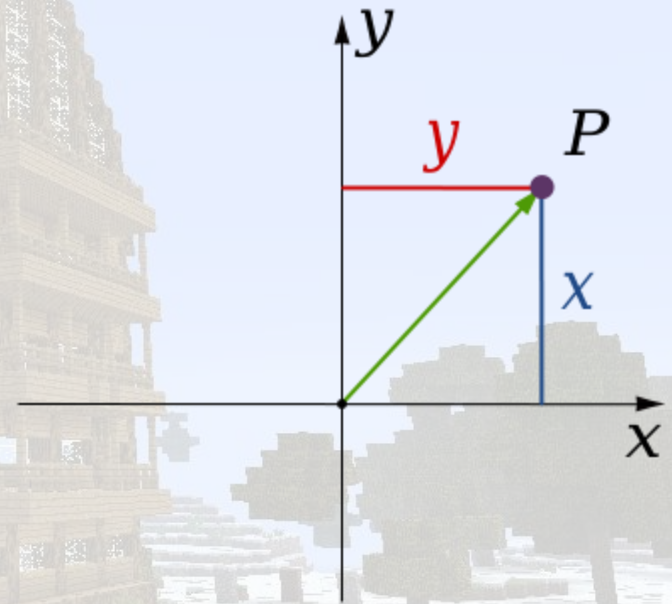


Components of a 3D game

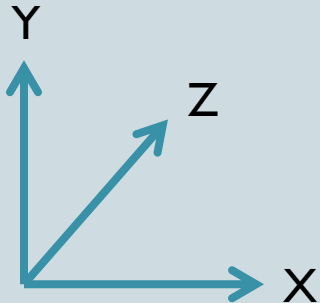
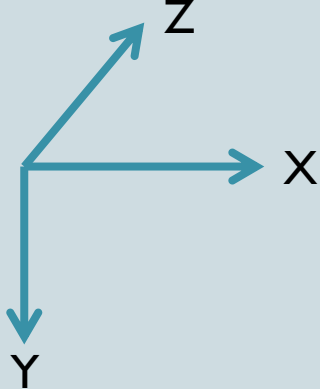
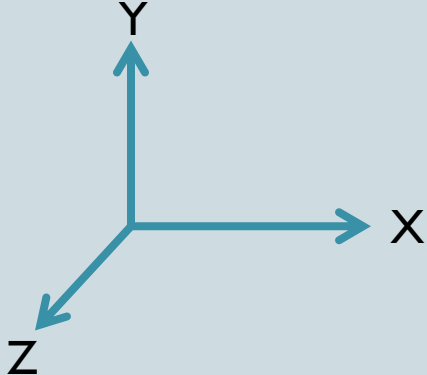


2D coordinate systems

- Standardized in the math world
- NOT standardized in the computer graphics world
- Sometimes, the same game engine may use different 2D coordinate systems for different purposes



3D coordinate systems

Left-handed	Modified left-handed	Right-handed
 A 3D coordinate system with three axes: X pointing to the right, Y pointing upwards, and Z pointing diagonally upwards and to the right. The Z-axis is perpendicular to the X-Y plane.	 A 3D coordinate system with three axes: X pointing to the right, Y pointing downwards, and Z pointing diagonally upwards and to the right. The Z-axis is perpendicular to the X-Y plane.	 A 3D coordinate system with three axes: X pointing to the right, Y pointing upwards, and Z pointing diagonally downwards and to the left. The Z-axis is perpendicular to the X-Y plane.
Used by DirectX (Windows / Xbox)	Used by Sony PlayStation	Used by OpenGL and Nintendo

Source: http://softimage.wiki.softimage.com/xwalkdocs/ftk_TemplateRef_SI_CoordinateSystem.htm

Basic terminology of 3D graphics

- **Matrix:** A rectangular array of numbers

For example: $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$ or (1) or $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$

- **Vector / point / column matrix:** Consists of 3 numbers, representing a point or a direction

For example $(1,2,3)$ or $\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$

- **Matrix multiplication:** When you multiply a 3x3 matrix by a vector, you get another vector.

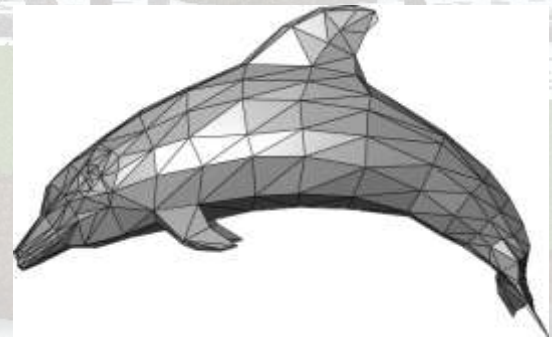
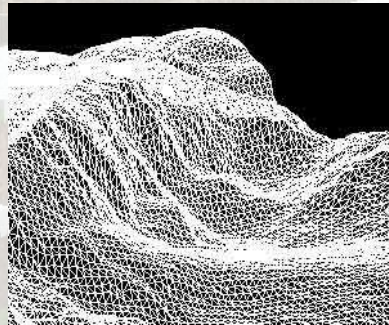
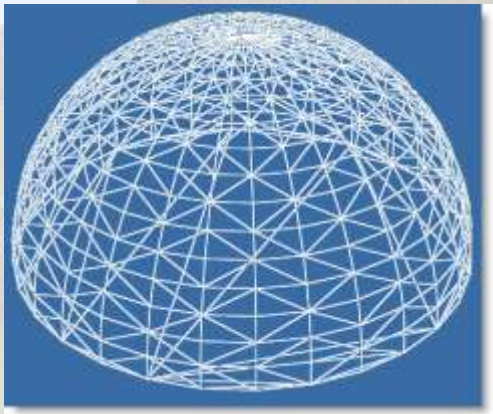
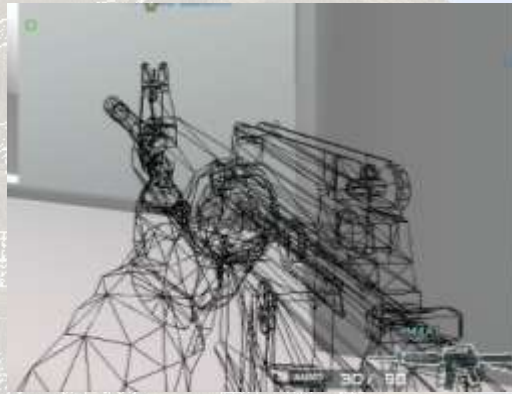
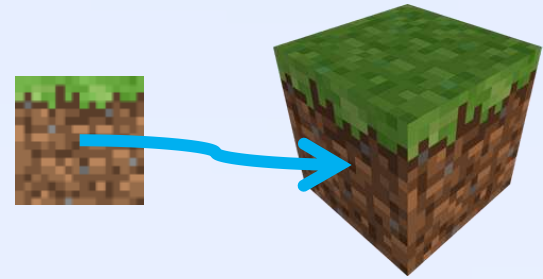
For example: $\begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \\ 6 \end{pmatrix}$

When you multiply a 3x3 matrix by another 3x3 matrix, you get another 3x3 matrix.

For example: $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$

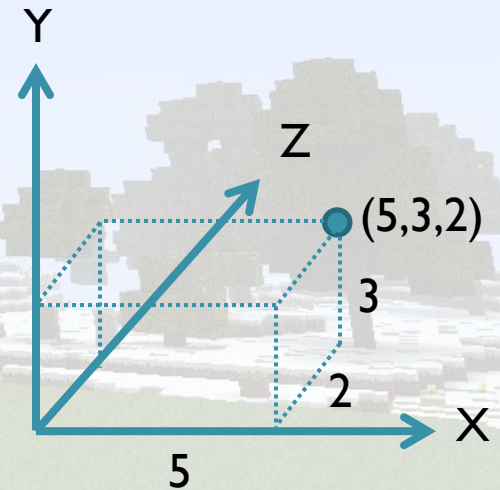
3D objects

- Every 3D object is basically made of 2 things:
 - Triangles
 - Zero or more 2D textures (images drawn onto the triangles)

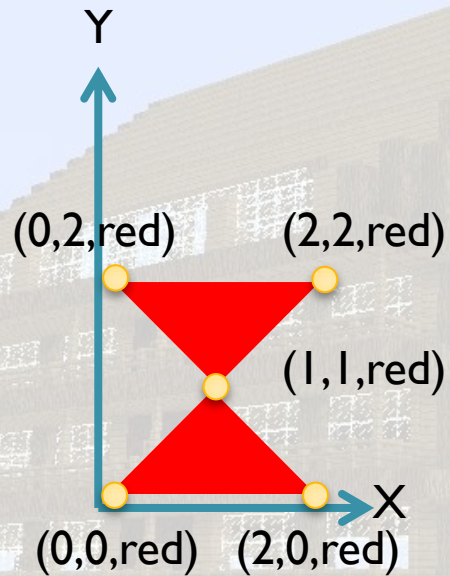


Vertices

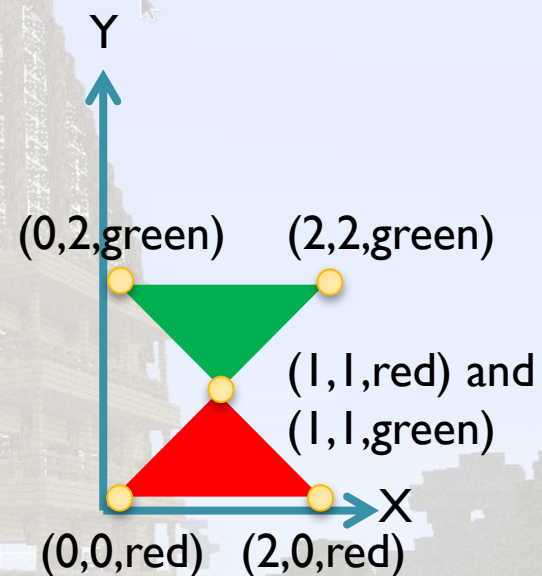
- A vertex is a point in space with some additional properties attached
- Basically, each vertex consists of:
 - Cartesian coordinates (x,y,z)
 - Color (a,r,g,b)
 - Texture coordinates (u,v)
- Objects can share vertices as long as these properties are the same



Vertex sharing - example

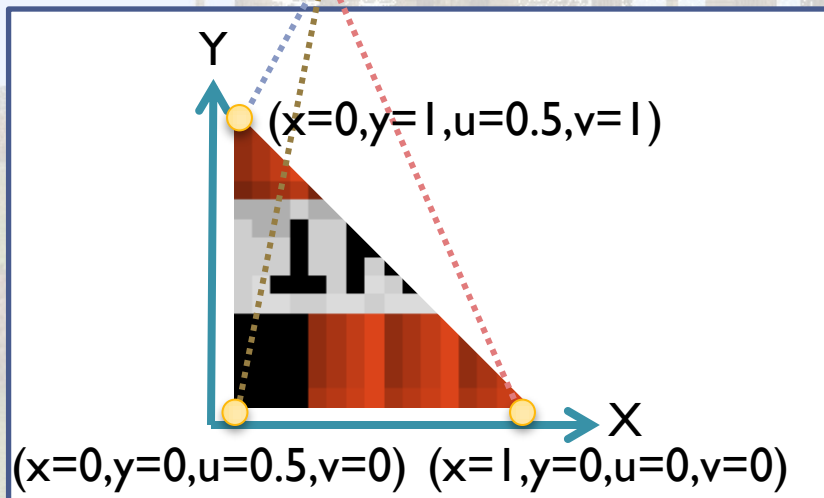


5 vertices



6 vertices

Textures



- A texture is an image drawn onto a triangle
- A single triangle can use only part of a texture
- Every vertex is assigned a texture coordinate (u, v)
- The texture coordinate tells the graphics card which part of the texture should be used to draw the triangle

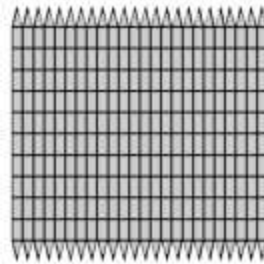
Textures

3-D Model



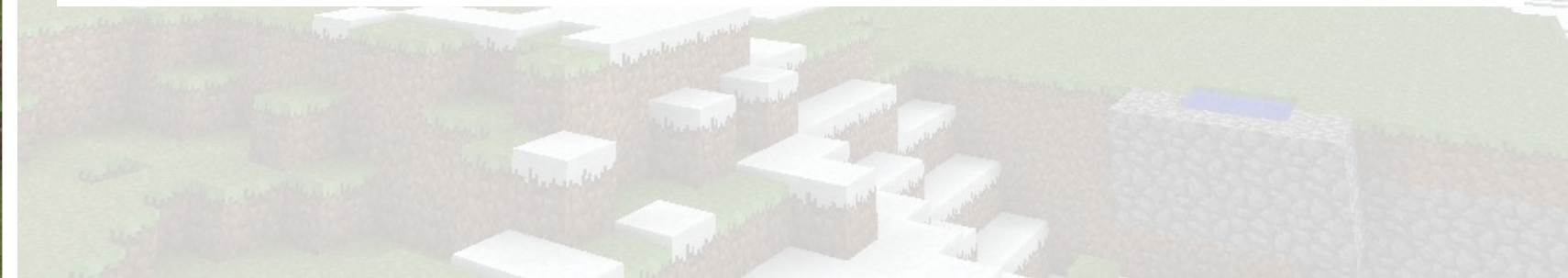
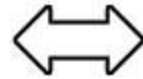
$$p = (x, y, z)$$

UV Map

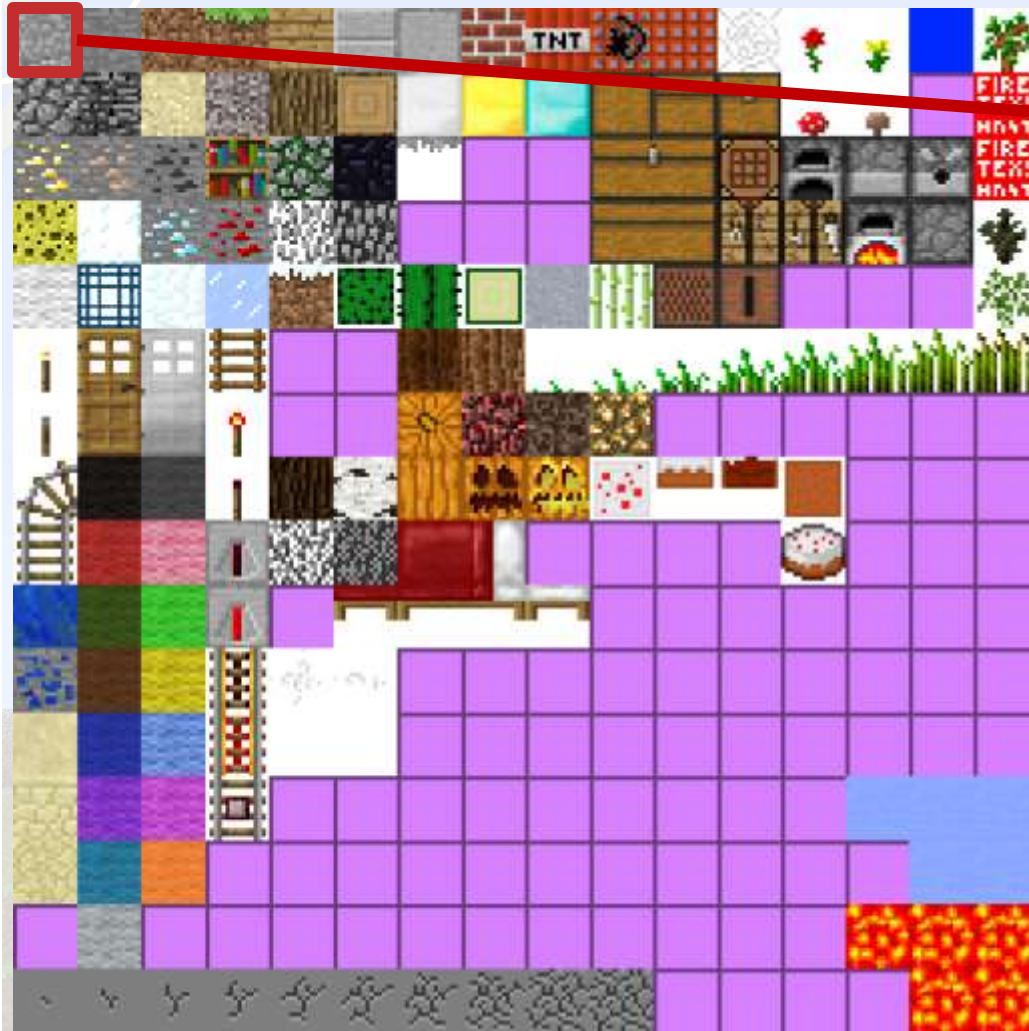


$$p = (u, v)$$

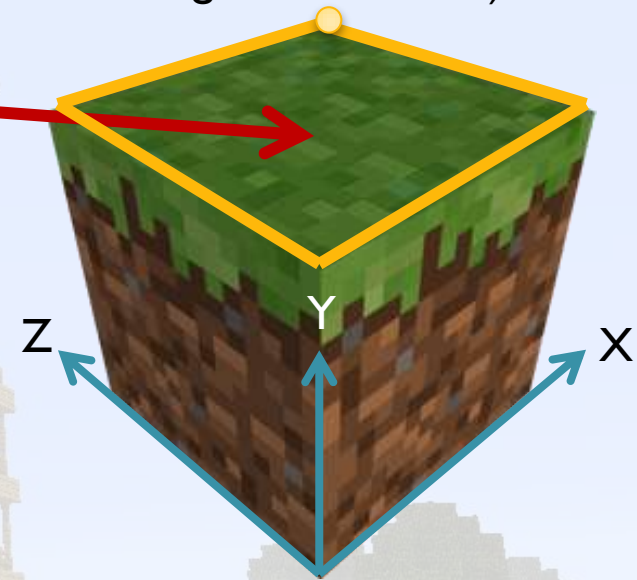
Texture



Textures



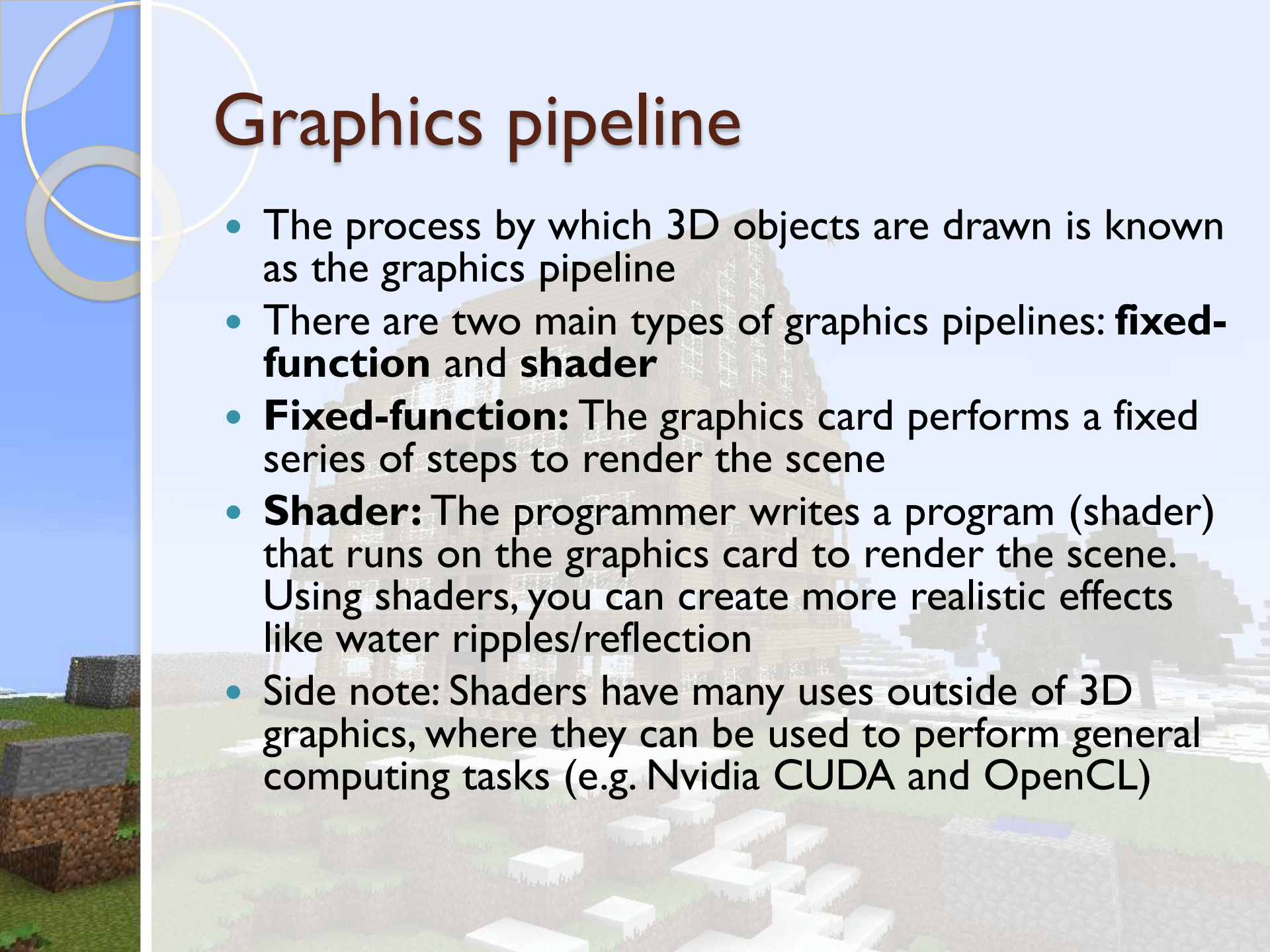
($x=1, y=1, z=1$,
color=green, $u=0, v=0$)



- Color of texture is combined with color of vertex
- This cube requires 24 vertices and 12 triangles



Graphics pipeline

- The process by which 3D objects are drawn is known as the graphics pipeline
 - There are two main types of graphics pipelines: **fixed-function** and **shader**
 - **Fixed-function:** The graphics card performs a fixed series of steps to render the scene
 - **Shader:** The programmer writes a program (shader) that runs on the graphics card to render the scene. Using shaders, you can create more realistic effects like water ripples/reflection
 - Side note: Shaders have many uses outside of 3D graphics, where they can be used to perform general computing tasks (e.g. Nvidia CUDA and OpenCL)
- 

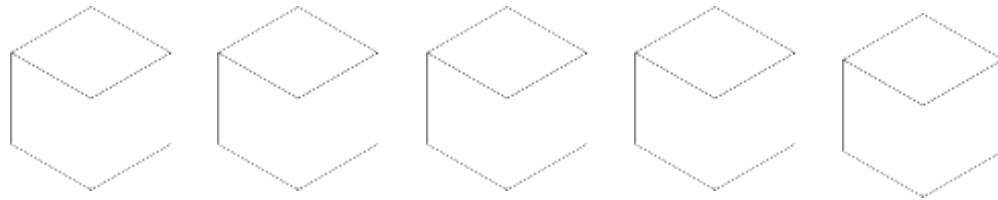
Fixed-function vertex pipeline

Simplified overview of steps

1. The vertices are multiplied by the **world matrix**, which transforms each vertex into its correct position
2. The vertices are multiplied by the **projection matrix**, which converts each 3D vertex into a 2D vertex to be displayed on the screen
3. Each triangle is rendered with its corresponding texture, using **depth testing** to determine which triangles are on top of others

World matrix

- Before rendering, each vertex is first transformed by the **world matrix** (also called the modelview matrix)
- The world matrix is supplied by the programmer and can be changed at any time
- By changing the world matrix, we can rotate, scale, and translate each point in the same way
- That way, we don't have to re-send all our vertices to the video card if we just want to transform all of them in the same way



- The same set of vertices can be used to draw all 5 cubes. Each cube is simply drawn with a different world matrix
- Only 8 vertices required, not 40

World matrix

- If the world matrix is the identity matrix, the vertices are not changed

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

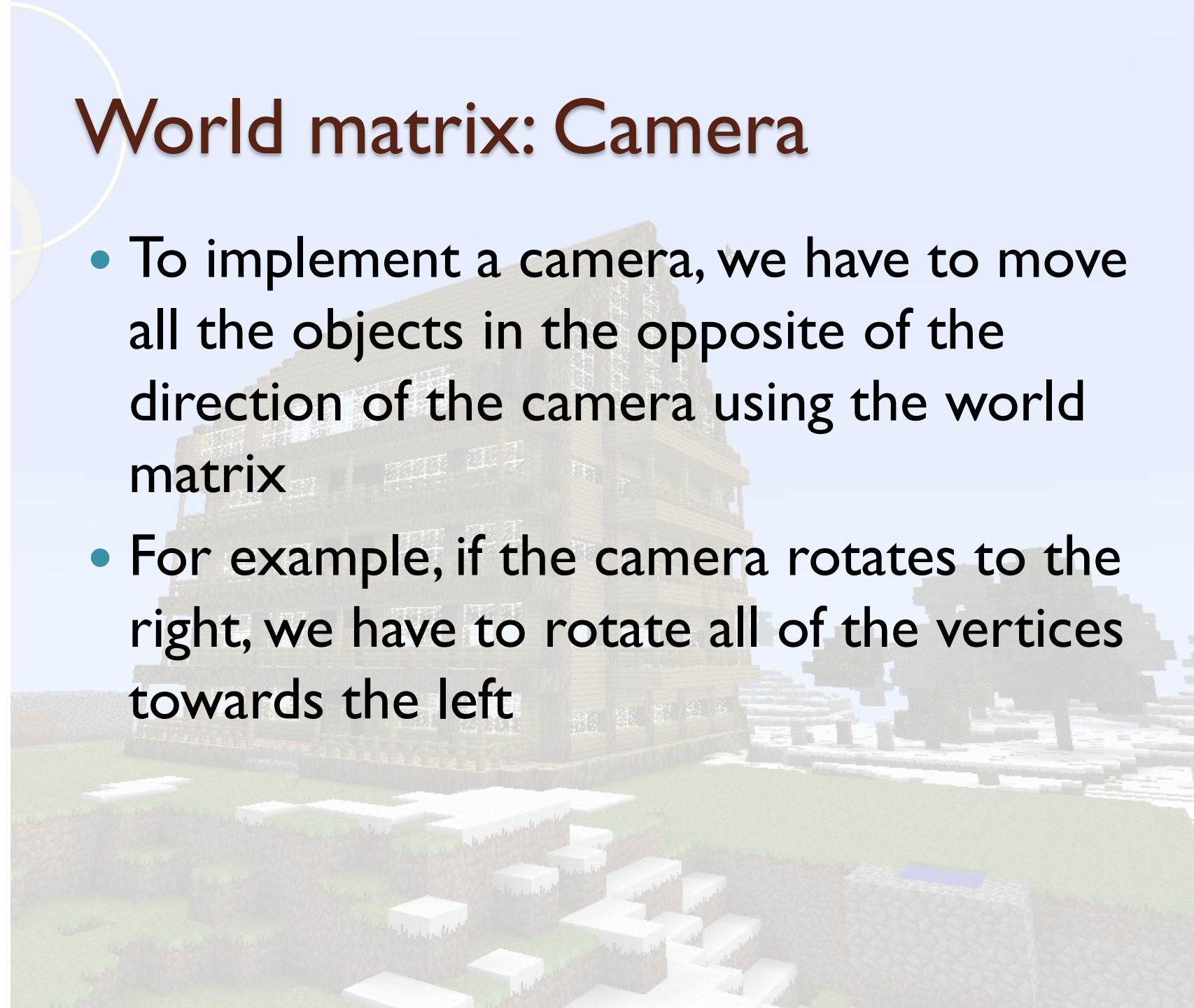
- Other matrices transform vertices in different ways

$$\begin{pmatrix} 0 & 0 & 4 \\ 0 & 3 & 0 \\ 2 & 0 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 2z \\ 3y \\ 4x \end{pmatrix}$$

- We can multiply several transformation matrices together to create a new transformation matrix. For example, if [A] rotates everything by 90 degrees about the X-axis and [B] translates everything 5 units in the negative Z direction, then [B][A] will rotate then translate.

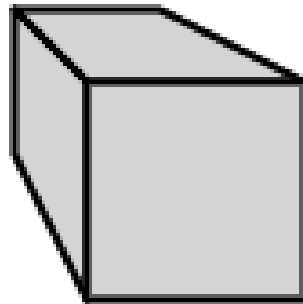
World matrix: Camera

- To implement a camera, we have to move all the objects in the opposite of the direction of the camera using the world matrix
- For example, if the camera rotates to the right, we have to rotate all of the vertices towards the left

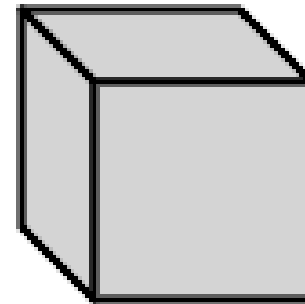


Projection matrix

- Next, each vertex is transformed by a **projection matrix**
- The projection matrix transform each 3D point into 2D points that can be displayed on the screen
- There are two main types of projection matrices, orthographic and perspective
- **Perspective:** Objects appear smaller when further away
- **Orthographic:** Objects are the same size no matter what



Perspective



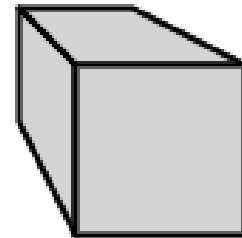
Orthographic

Projection matrix

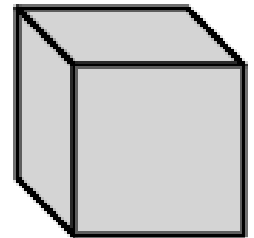
- Orthographic projection – z coordinate is thrown out the window

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x \\ y \\ 0 \end{pmatrix}$$

- Perspective projection – more complicated (involves additional coordinate)



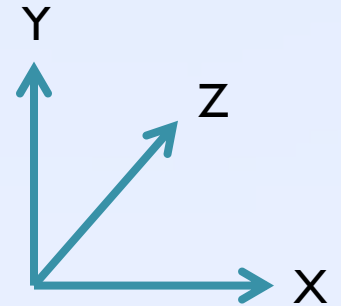
Perspective



Orthographic

Depth testing

- How does the graphics card decide which triangles are hidden by others?
- The most common solution is called depth testing (also known as z-buffering)
- For every frame, the graphics card must keep track of the nearest seen Z value for every single pixel on the screen
- When a triangle is drawn, the Z value of each pixel is compared against the value in the Z buffer and will only be drawn if it is nearer



Depth testing

