CMSC427: Computer Graphics
Lecture Notes
Last update: November 21, 2014

TA: Josh Bradley

1 Linear Algebra Review

1.1 Vector Multiplication

Suppose we have a vector $\vec{\mathbf{a}} = \begin{bmatrix} x_a & y_a & z_a \end{bmatrix}^T$. Then for some scalar c,

$$c\vec{\mathbf{a}} = \begin{bmatrix} c \times x_a \\ c \times y_a \\ c \times z_a \end{bmatrix}$$

This idea of each component of a vector getting multiplied by the scalar extends to vectors of arbitrary size.

1.2 Vector Division

Although there are solutions to problems where the idea of vector division can be applied, in general vector division cannot be uniquely defined in terms of matrices.

1.3 Vector Addition

Suppose there are two vectors $\vec{\mathbf{a}} = \begin{bmatrix} x_a & y_a & z_a \end{bmatrix}^T$ and $\vec{\mathbf{b}} = \begin{bmatrix} x_b & y_b & z_b \end{bmatrix}^T$. Then we define vector addition as the sum of each individual component.

$$\vec{\mathbf{a}} + \vec{\mathbf{b}} = \begin{bmatrix} x_a \\ y_a \\ z_a \end{bmatrix} + \begin{bmatrix} x_b \\ y_b \\ z_b \end{bmatrix} = \begin{bmatrix} x_a + x_b \\ y_a + y_b \\ z_a + z_b \end{bmatrix}$$

Vector addition extends to vectors with an arbitrary number of components. For vector addition to work, both vectors must be of equal size.

1.4 Vector Subtraction

Vector subtraction is the same as vector addition, but with negated components. Suppose we define two vectors $\vec{\mathbf{a}} = \begin{bmatrix} x_a & y_a & z_a \end{bmatrix}^T$ and $\vec{\mathbf{b}} = \begin{bmatrix} x_b & y_b & z_b \end{bmatrix}^T$. Then we define vector subtraction as the difference between each individual component.

$$\vec{\mathbf{a}} - \vec{\mathbf{b}} = \begin{bmatrix} x_a \\ y_a \\ z_a \end{bmatrix} - \begin{bmatrix} x_b \\ y_b \\ z_b \end{bmatrix} = \begin{bmatrix} x_a - x_b \\ y_a - y_b \\ z_a - z_b \end{bmatrix}$$

Vector subtraction extends to vectors with an arbitrary number of components. Both vectors must be of equal size.

1.5 Matrix Multiplication

In general, given two matrices $\mathbf{A}$ and $\mathbf{B}$,

$$\mathbf{AB} \neq \mathbf{BA}$$

If $\mathbf{A}$ is size $m \times n$ and $\mathbf{B}$ is size $p \times q$, then $\mathbf{AB}$ only works if $n = p$. The resulting matrix will have size $m \times q$.

1.6 Determinant

The determinant is a value associated with a square matrix. There are multiple ways to calculate the determinant. Let I_n be the $n \times n$ identity matrix and $\mathbf{A}$ be a matrix. Properties of the determinant include:

$$\begin{aligned} det(I_n) &= 1 \\ det(A^T) &= det(A) \\ det(A^{-1}) &= \frac{1}{det(A)} \\ det(cA) &= c^n det(A) \text{ for a } n \times n \text{ matrix} \end{aligned}$$

If A and B are square matrices of equal size, then

$$det(AB) = det(A)det(B)$$

2 Barycentric Coordinates

2.1 2D Triangles

The barycentric coordinate system is a coordinate system where the location of an arbitrary point inside some triangle is specified as the center of mass (a.k.a barycenter) of masses placed at the vertices.

Given some triangle where the vertices are labeled in order counter-clockwise (see figure below), let us assume the coordinate origin is $\mathbf{a}$ and the vectors from $\mathbf{a}$ to $\mathbf{b}$ and $\mathbf{a}$ to $\mathbf{c}$ are basis vectors. Using $\mathbf{a}$ as the origin and these two vectors as basis vectors, any point $\mathbf{p}$ can be written as as combination of these vectors.

$$\mathbf{p} = \alpha + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a})$$

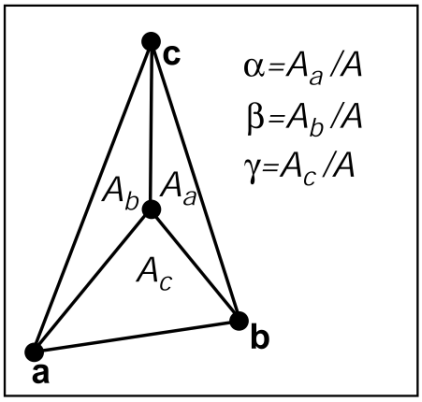
under the constraint that

$$\alpha + \beta + \gamma = 1$$

A nice feature of barycentric coordinates is that a point $\mathbf{p}$ is inside the triangle formed by $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ if and only if

$$\begin{aligned} 0 < &\alpha < 1 \\ 0 < &\beta < 1 \\ 0 < &\gamma < 1 \end{aligned}$$

If one of these coordinates is 0, this means the point $\mathbf{p}$ is on an edge of the triangle.



One way to compute barycentric coordinates is to look at the proportion of the area of the subtriangles formed by the point $\mathbf{p}$. If we let A_a, A_b, and A_c represent the area of their respective subtriangles and A is the area of the whole triangle, the barycentric coordinates have the following relationship with the subtriangle areas:

$$\alpha = \frac{A_a}{A}$$
$$\beta = \frac{A_b}{A}$$
$$\gamma = \frac{A_c}{A}$$

2.2 3D Triangles

Barycentric coordinates easily extend to 3D. When discussing triangles in 3D, we must first know how to compute the normal vector. The normal vector is found by taking the cross product of any two vectors in the triangle. Since we're talking about triangles, the best thing to do is choose two vectors corresponding to two of the edges of the triangle, therefore we let

$$\mathbf{n} = (\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a})$$

The whole triangle area can then be computed from this normal vector as

$$\text{area} = \frac{1}{2}\|\mathbf{n}\| = \frac{1}{2}\|(\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a})\|$$

Barycentric coordinates cannot be directly computed from this area though, because it does not give you a *signed* area. Using the above equations, we can however derive equations that will compute the barycentric coordinates. These derivations result in the following equations for the barycentric coordinates:

$$\alpha = \frac{\mathbf{n} \cdot \mathbf{n_a}}{\|\mathbf{n}\|^2}$$
$$\beta = \frac{\mathbf{n} \cdot \mathbf{n_b}}{\|\mathbf{n}\|^2}$$
$$\gamma = \frac{\mathbf{n} \cdot \mathbf{n_c}}{\|\mathbf{n}\|^2}$$

where

$$\mathbf{n_a} = (\mathbf{c} - \mathbf{b}) \times (\mathbf{p} - \mathbf{b})$$
$$\mathbf{n_b} = (\mathbf{a} - \mathbf{c}) \times (\mathbf{p} - \mathbf{c})$$
$$\mathbf{n_c} = (\mathbf{b} - \mathbf{a}) \times (\mathbf{p} - \mathbf{a})$$

3 Transformation Matrices

In computer graphics, we can use matrix operations to perform various image operations. These include scaling, rotation, and shearing. Shearing is the visual equivalent of slanting in an image. See `http://en.wikipedia.org/wiki/Transformation_matrix#Examples_in_2D_graphics` to see examples of what the matrix for each of these operations look like.

3.1 2D Transformations

3.1.1 Translation

In 2D, we typically see transformation matrices of the form

$$\begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_1' \\ x_2' \end{bmatrix}$$

However it is not possible to perform a translation operation as a result of matrix multiplication in this form. For a translation, we want a matrix multiplication that results in the following equations

$$x_1 + c_1 = x_1'$$
$$x_2 + c_2 = x_2'$$

To achieve this, we need to do a little "trick". We express the 2D transformation as a transformation in 3D. This gives us the following matrix:

$$\begin{bmatrix} 1 & 0 & c_1 \\ 0 & 1 & c_2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix} = \begin{bmatrix} x_1 + c_1 \\ x_2 + c_2 \\ 1 \end{bmatrix} = \begin{bmatrix} x_1' \\ x_2' \\ 1 \end{bmatrix}$$

For the purposes of a translation in 2D, the last element of the resulting vector is normally ignored.

3.1.2 Rotation

In 2D, we can express an anti-clockwise rotation around the origin as a 2x2 matrix.

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

3.1.3 Scaling

In 2D, we can express a scaling operation as a 2x2 matrix.

$$\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x * s_x + 0 \\ 0 + y * s_y \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

3.2 3D Transformations

3.2.1 Translation

In 3D, we can express a translation operation as a 4x4 matrix.

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix}$$

3.2.2 Rotation

In 3D, rotation can be expressed as a collection of three different 4x4 matrices (one for each dimension). We must first define 4x4 anti-clockwise rotation matrices for each direction.

$$R_x(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Then to do a complete rotation in 3D, you must know the degrees, θ_x, θ_y, θ_z, to rotate anti-clockwise in each direction. You can then multiple the three matrices together.

$$R = R_z(\theta_z)R_y(\theta_y)R_x(\theta_x)$$

From here, we can then multiply the resulting rotation matrix with a point in 3D space.

$$R \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix}$$

3.2.3 Scaling

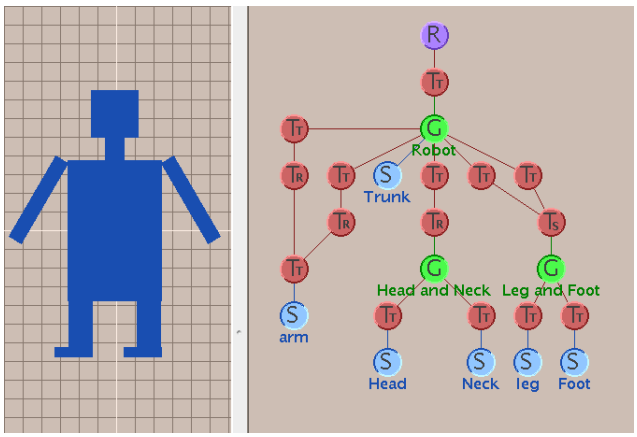
In 3D, we can express scaling as a 4x4 matrix.

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x * s_x \\ y * s_y \\ z * s_z \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix}$$

4 Scene Graph Data Structure

A scene graph is a tree-like data structure that contains the logical and (usually) spatial representation of a graphical scene. Scene graphs can take on many different forms and the nodes can represent different things. For example, if we wanted to create a graph for a robot, then the scene graph data structure might look like the one below. Each blue node in the tree contains data pertaining to the coordinates of a piece of the robot. Green nodes are used to group together parts (blue nodes) of the robot that will move together. For example, any transformation that is applied to the head will also get applied to the neck in this example. More advanced scene graphs also contain information related to the fragment shader for each piece so that each piece can be colored differently. The red nodes in the graph represent the *types* of transformations that are possible in the scene graph. These are optional nodes, because they restrict the scene graph to performing only certain operations on parts of the robot. In the following example, we use the following notation

$$T_t = \text{transformation matrix}$$
$$T_s = \text{scaling matrix}$$
$$T_r = \text{rotation matrix}$$



5 Meshes

5.1 Artifacts

There are several ”artifacts” or ”things to consider” when working with meshes. They include

- holes
- isles
- inconsistent orientation
- large scale overlap
- complex angles
- intersection

5.2 Data Structures

There are several kinds of data structures used to represent meshes. For this class, we will focus on Face-Vertex meshes. A Face-Vertex mesh is comprised of the following two data structures:

1. Face List - a list of triangle faces where each triangle face is denoted as a list of three vertices making up the 2D triangle face.
2. Vertex List - a list of all vertices being used. Each vertex is made up of coordinates corresponding to the space it lives in (i.e. 2D, 3D).

5.3 Local Operations on Meshes

There are several *local* operations that are performed on a mesh. We call these operations *local* because they affect only one part of the mesh, and not the entire mesh. Examples of these operations include

- Edge Swap - used to get rid of short skinny triangles
- Edge Collapse - useful for mesh simplification
- Triangle Collapse - removing an entire triangle from the mesh
- Vertex Removal - after removing a vertex, multiple faces are lost Retriangulation is needed afterwards in order to guarantee all faces are still triangles

6 Shading

When discussing shading, it is important to first introduce the idea of a light ”source”. A light source is modelled as a some point in 3D space in the model. A light source is necessary before we can start doing things like finding shadow areas and performing the appropriate shading.

The amount of light reaching a surface depends on the orientation of the surface to the light source. Points on a surface that are ”closer” to a light source and not being blocked will be brighter than a point far away from the surface that is not blocked.

6.1 Phong Shading

Phong shading is an interpolation technique used in shading a surface in 3D. It is sometimes called normal-vector interpolation shading.

6.2 Diffuse Shading

Objects that have "matte" appearance. Light is scattered in all direction equally. The intensity at any point doesn't depend on the angle of the viewer. The amount of light reaching the surface depends on the orientation of the surface to the light source.

6.3 Extra Notes

Shiny models will have bright spots, which we call specularities. These specularities are the mirror-like reflection of light on a surface.

7 Quaternions

Quaternions extend the idea of complex numbers. Quaternions are especially useful in computer graphics to perform 3D rotations. To define a quaternion, we first review complex numbers.

7.1 Complex Numbers Review

All complex numbers $z \in \mathbb{C}$ can be written in the following form where $x, y \in \mathbb{R}$ are real numbers:

$$z = x + yi$$

The x is the real part and y is the imaginary part. Below is a look at what the complex numbers look like in the complex plane. Note that complex numbers are often written in the form of polar coordinates instead of euclidean coordinates (which we are familiar with the most). We will soon see why this is done.

7.1.1 Addition

Complex number addition just requires adding the parts of each complex number together

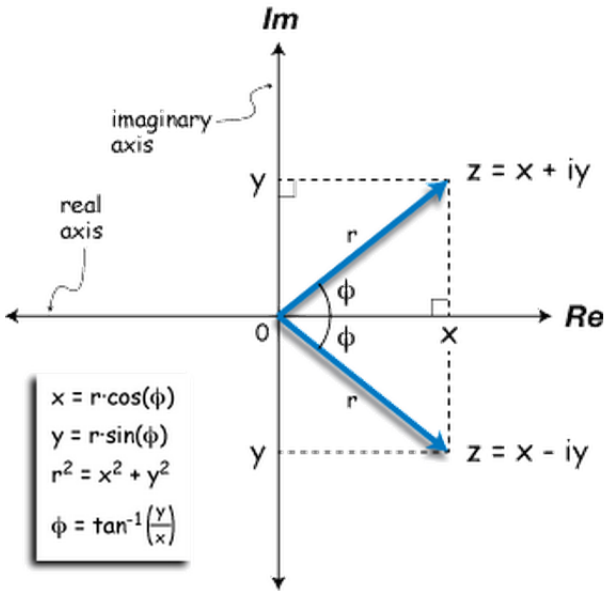
$$(x_1 + y_1 i) + (x_2 + y_2 i) = (x_1 + x_2) + (y_1 + y_2)i$$

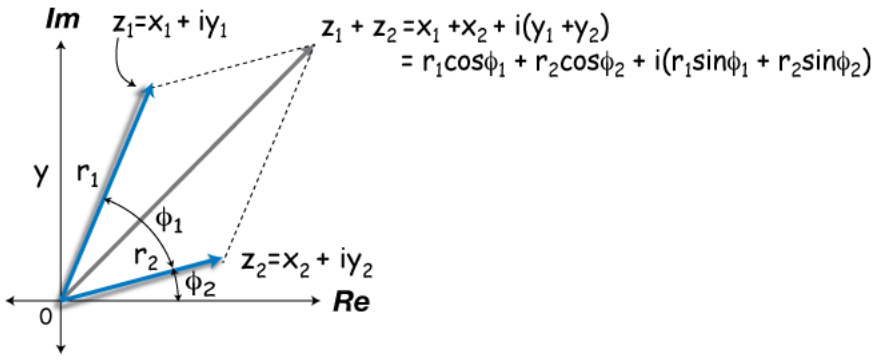
Graphically, complex number addition is the same as vector addition (see below).

7.1.2 Subtraction

Similar to addition, complex number subtraction requires subtracting the parts of each complex number together

$$(x_1 + y_1 i) - (x_2 + y_2 i) = (x_1 - x_2) + (y_1 - y_2)i$$





7.1.3 Multiplication

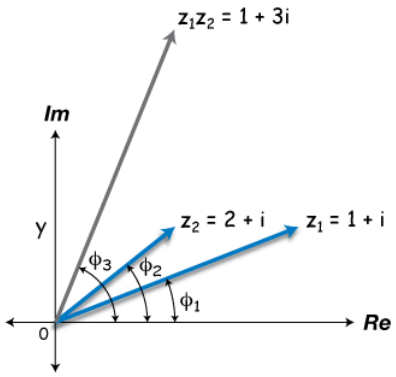
To multiply two complex numbers, we just use the standard FOIL method of multiplying numbers inside parentheses and use the complex number property $i^2 = -1$

$$\begin{aligned}(x_1 + y_1 i)(x_2 + y_2 i) &= x_1 x_2 + x_1 y_2 i + y_1 x_2 i + y_1 y_2 i^2 \\ &= (x_1 x_2 - y_1 y_2) + (x_1 y_2 + y_1 x_2) i\end{aligned}$$

Graphically speak, multiplication of two complex numbers performs a rotation around the origin, as well as modifies the length of the resulting vector (i.e. complex number).

7.2 Definition

A quaternion is made up of two components that represent two different things. One component is a x, y, z point in 3D space that represents a vector (axis) about which a rotation will occur. The second component is some number w that represents the amount of rotation to occur about the axis specified in the first component. Therefore, a quaternion is often written in the form



$$q = w + xi + yi + zi$$

where i is the imaginary number. For the sake of simplicity, we can drop the i from the above equation. The imaginary component is important when trying to understand the math behind quaternions. Now, we can simplify the description of a quaternion and say that w is the amount of rotation about the axis defined by (x, y, z).

7.3 Properties

In graphics, we are typically only concerned with quaternions that have unit length. In fact, quaternions are often *assumed* to be of unit length. To calculate the unit quaternion (or to just check and make sure it has unit length), we can normalize the quaternion like we would with any vector.

$$\begin{aligned} \text{Let} \quad & |q| = \sqrt{w^2 + x^2 + y^2 + z^2} \\ \text{Then} \quad & w = w/|q| \\ & x = x/|q| \\ & y = y/|q| \\ & z = z/|q| \end{aligned}$$

When two unit quaternions are multiplied together, their product (called the Hamilton product) is a quaternion that also has unit length!. Let $Q_1 = w_1 + x_1 + y_1 + z_1$ and $Q_2 = w_2 + x_2 + y_2 + z_2$ be two quaternions and suppose $Q_3 = (Q_1 * Q_2)$ is the resulting quaternion. Then, quaternion multiplication is just like vector multiplication, with the only exception being the inclusion of the imaginary number. The following formulas can be derived using the complex property $i = \sqrt{-1}$:

$$\begin{aligned} Q_3.w &= (Q_1 * Q_2).w = (w_1w_2 - x_1x_2 - y_1y_2 - z_1z_2) \\ Q_3.x &= (Q_1 * Q_2).x = (w_1x_1 + x_1w_2 + y_1z_2 - z_1y_2) \\ Q_3.y &= (Q_1 * Q_2).y = (w_1y_2 - x_1z_2 + y_1w_2 + z_1x_2) \\ Q_3.z &= (Q_1 * Q_2).z = (w_1z_2 + x_1y_2 - y_1x_2 + z_1w_2) \end{aligned}$$

One important thing to remember is that unlike vector multiplication, quaternion multiplication is not commutative, therefore $(Q_1 * Q_2) \neq (Q_2 * Q_1)$.

7.4 Quaternion Rotation

Suppose there is a vector p and quaternion q. Then we can transform p into a quaternion by setting the axis component of the quaternion equal to p and setting the rotation component to 0. In other words, we define a new quaternion p_q as

$$p_q = (w, x, y, z) = (0, p_x, p_y, p_z)$$

Then a 3D rotation of p_q by q is given by

$$p' = qp_qq^{-1}$$

This equation simplifies down to a simpler form though

$$p' = p_q + 2w(v \times p_q) + 2(v \times (v \times p_q))$$

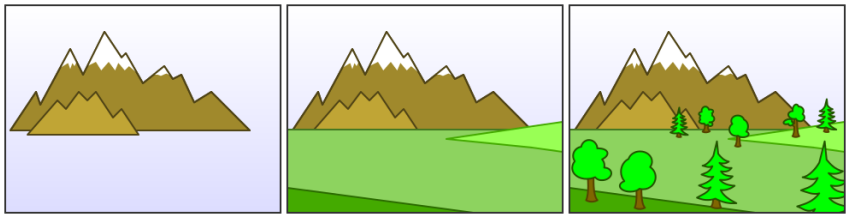
where v is the (x, y, z) component of q.

8 Depth

In graphics, when rendering 3D models, you will inevitably encounter the *visibility problem*, which is the problem of deciding which elements of a rendered scene are visible, and which are hidden. We will talk about two solutions to this problem.

8.1 Painter's Algorithm

The painter's algorithm is based off the simple idea of drawing (a.k.a. painting) objects *farthest* away first. In the mountain scene below, the mountains are painted first, followed by the meadow, and then the trees. Although some trees are more distant from our viewpoint than parts of the meadow, the ordering (mountains, meadow, trees) forms a valid depth order. The main idea behind the Painter's algorithm is that no object in the ordering obscures any part of a later object.



8.2 Z-Buffering

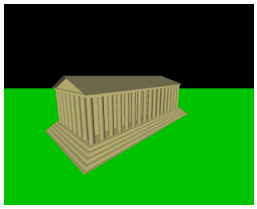
Z-buffering is another solution to the *visibility problem*. It is the most widely used solution in contemporary computers, laptops and mobile phones for performing 3D graphics. This method gets its name from the fact that a buffer is used to maintain depth information for each pixel on a 2D computer screen.

A Z-buffer is normally represented as a 2D array where each (x,y) element in the buffer corresponds to a screen pixel. The value of each element in the Z-buffer represents the *depth* of that pixel. If an object of a

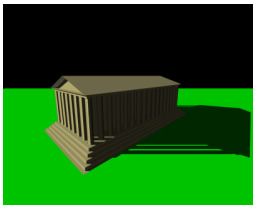
scene must be rendered in the same pixel as another object that is already there (i.e. overlapping objects), the algorithm compares the two depths of the objects and overrides the current pixel if the object is closer to the observer. The chosen depth of the pixel is then saved back to the z-buffer, replacing the old one. After everything is done, the final Z-buffer will allow the method to correctly reproduce the depth perception we expect: a close object hides a farther one.

9 Shadows

In 3D scenes, whenever a light source is used, most people expect there to be shadows to appear because that is what happens in the real world. Otherwise, a scene can appear to be fake (image a game with no shadows). The technique by which shadows are added to 3D computer graphics is called *shadow mapping* or *projective shadowing*. The idea of shadow mapping was first introduced by Lance Williams in 1978.



No shadows

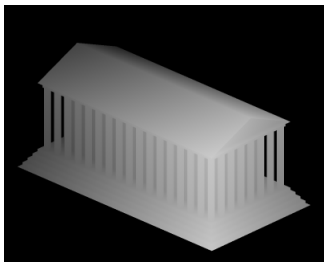


With shadows

9.1 Algorithm

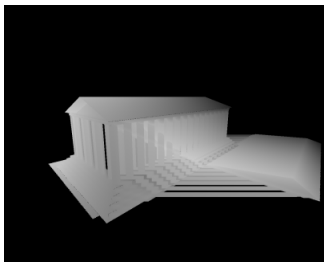
Rendering a scene with shadows (like the one below) requires two major drawing passes over a scene.

1. In the first pass, the shadow map will get produced. In the past, we have always calculated everything from the view of the camera. In other words, the camera had a ”view frustum” associated with it and everything was calculated from the viewpoint of the camera. To produce a ”shadow map” though, we assume the light source has a view frustum and render the scene from the viewpoint of the light source instead. In this pass though, only the depth map should be saved.The depth map is the ”shadow map”. The scene from the point of view of the light source is sometimes called ***light space***. The shadow map for the above scene is:



Depth map created from the viewpoint of the light source

2. In the second pass, we must render the scene as usual (from camera’s viewpoint) but apply the shadow map by projecting it onto the scene. In this pass, there are three steps that must be followed:



Depth map projected onto the scene from the viewpoint of the camera

(a) Starting with a coordinate in the scene (from the camera's viewpoint), find its corresponding coordinate in light space by using a matrix transformation. The matrix responsible for transforming the coordinates to the lights viewing coordinates is the product of the modelview and projection matrices.

(b) Compare the coordinate (from the camera's viewpoiunt) with its corresponding value in the depth map.

(c) Based on the comparison, draw the object either in shadow or light.

For more details on this process, please visit `http://en.wikipedia.org/wiki/Shadow_mapping#Shading_the_scene`