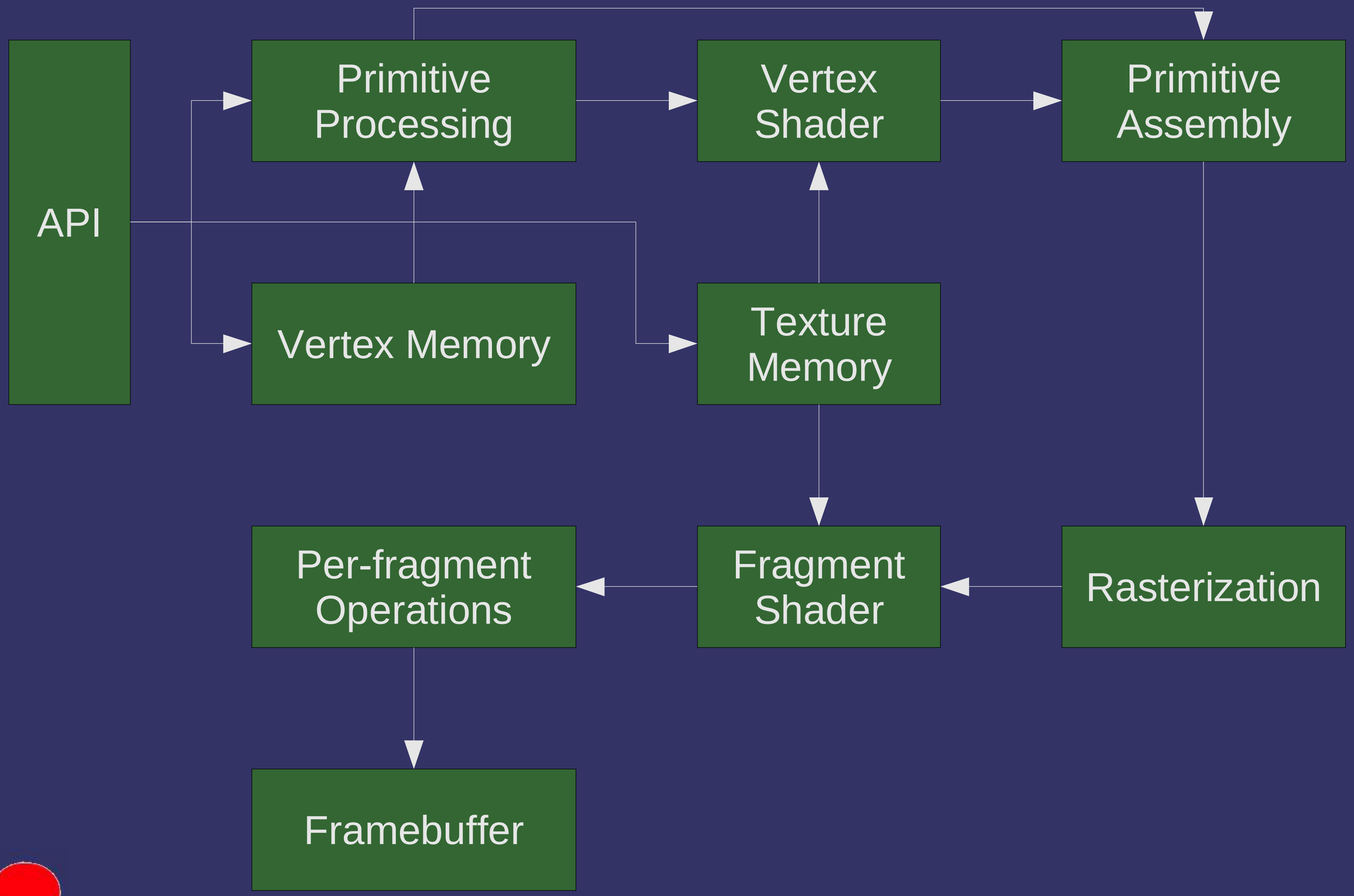
# *VGP351 – Week 2*

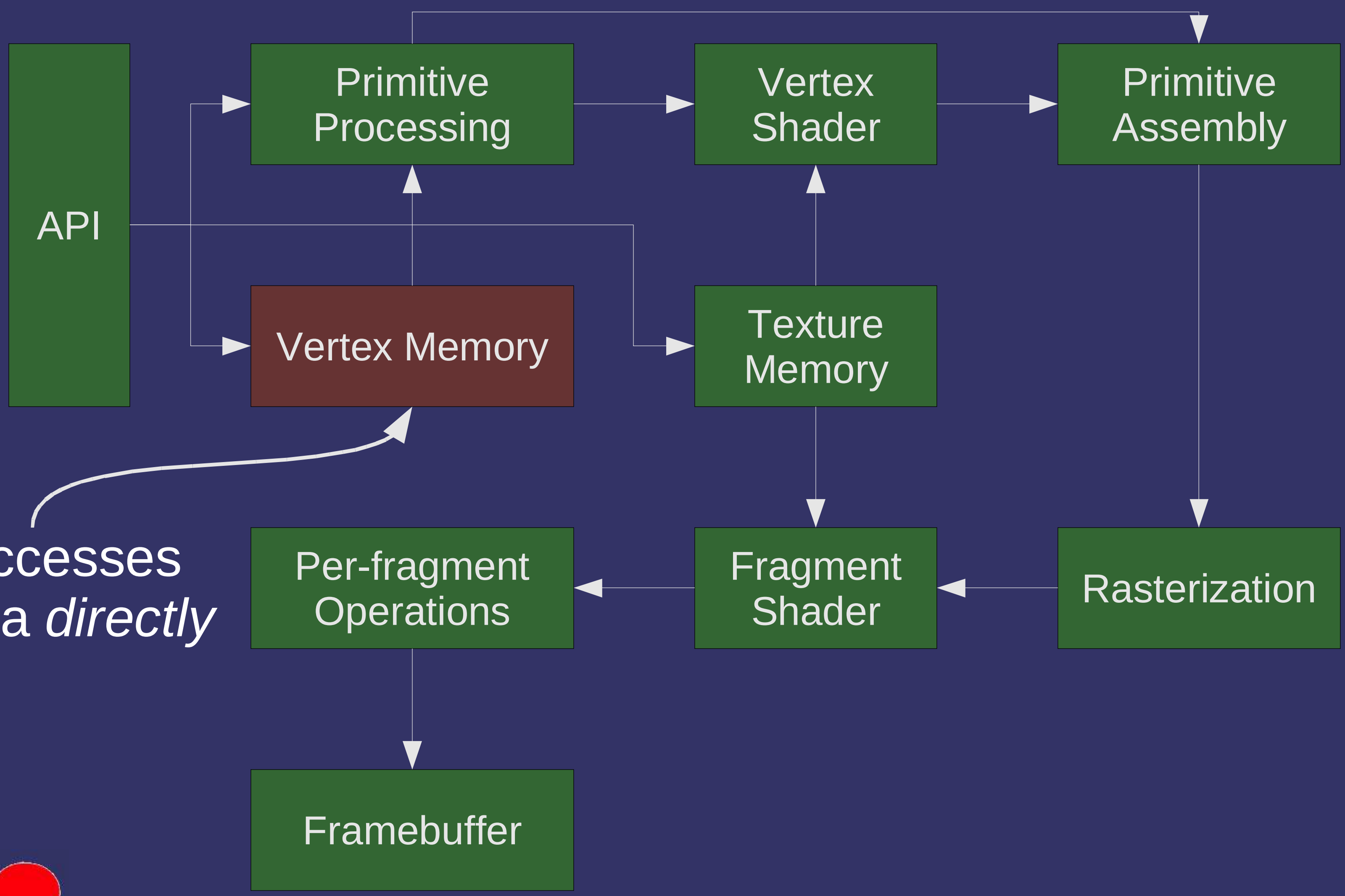
- Agenda:
  - Getting data to the GPU
    - Buffer objects
    - Vertex attributes
    - Uniforms
  - Types of primitives
  - Transformations
    - Modeling
    - Viewing
    - Projection

12-October-2011

# Graphics Pipeline



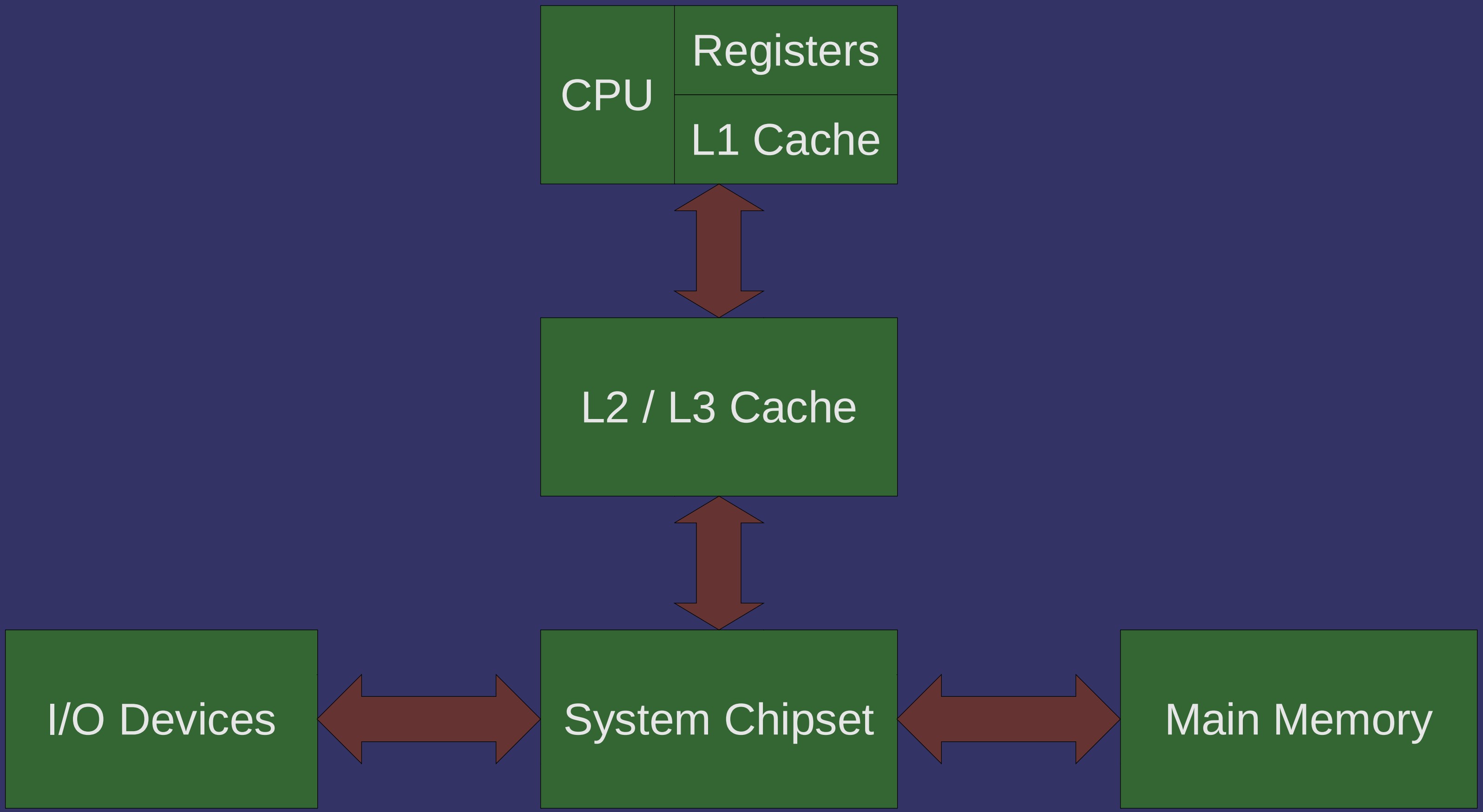
# *Graphics Pipeline*




GPU accesses
this data *directly*

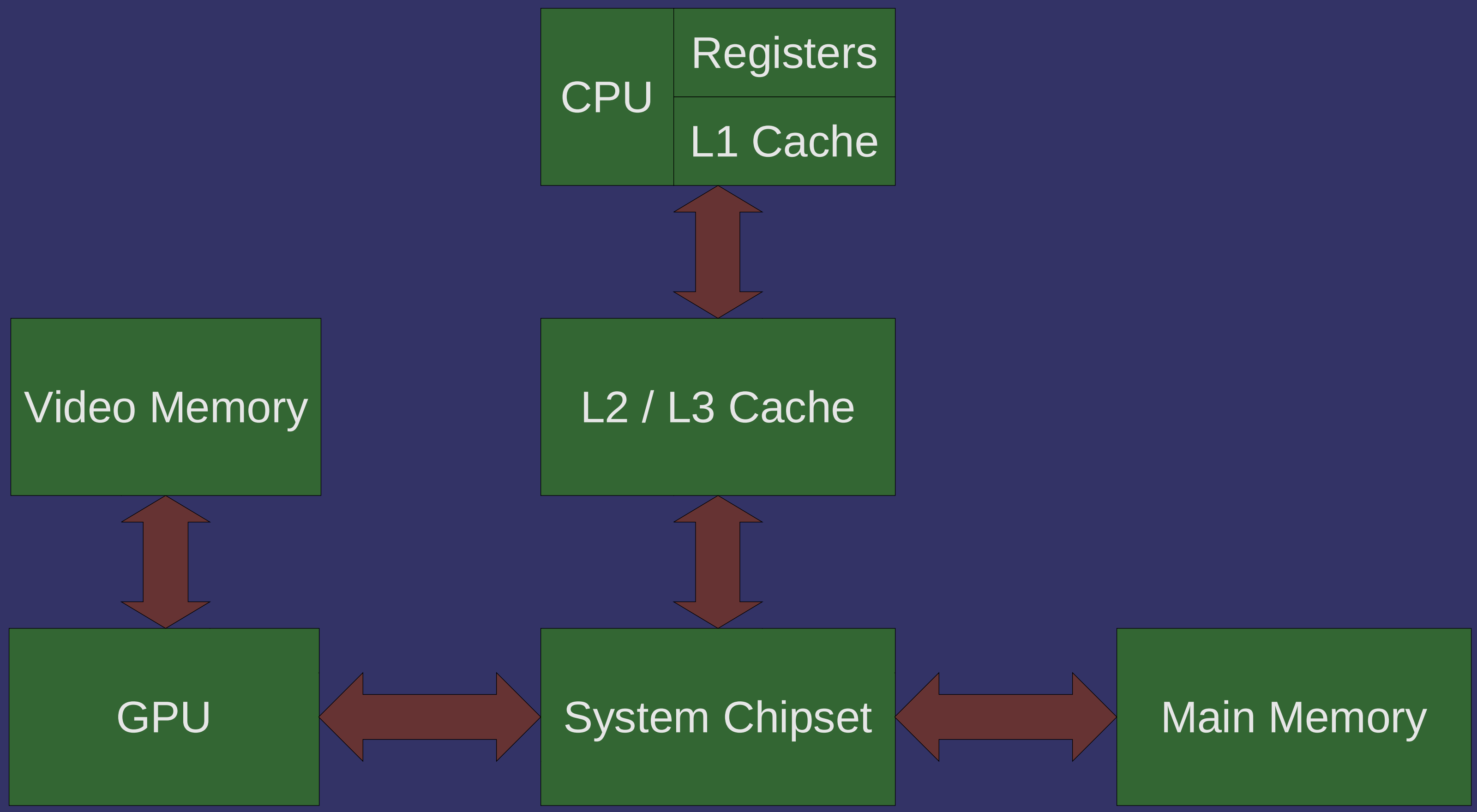
12-October-2011

# *Memory Architecture*



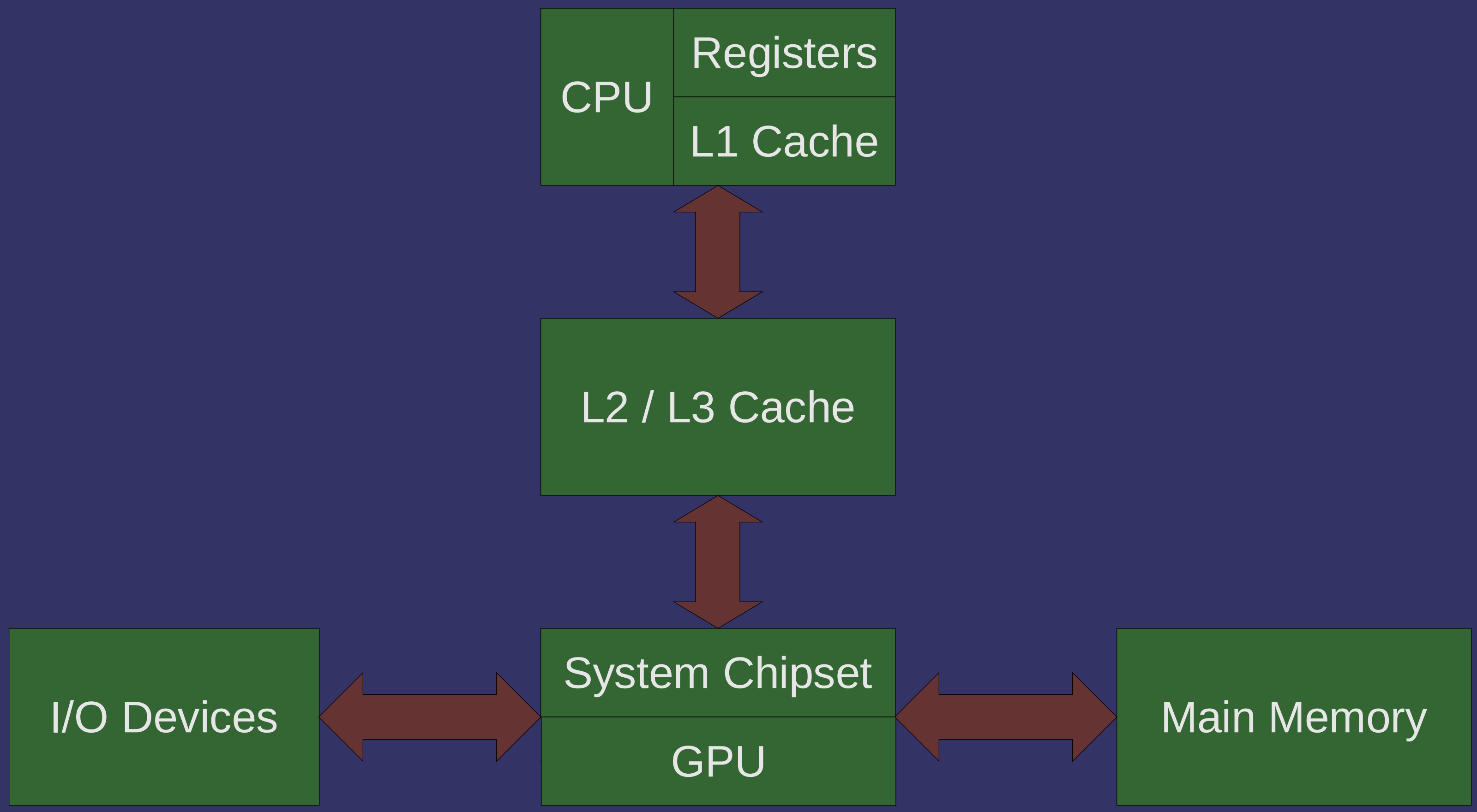

12-October-2011

# *Memory Architecture*

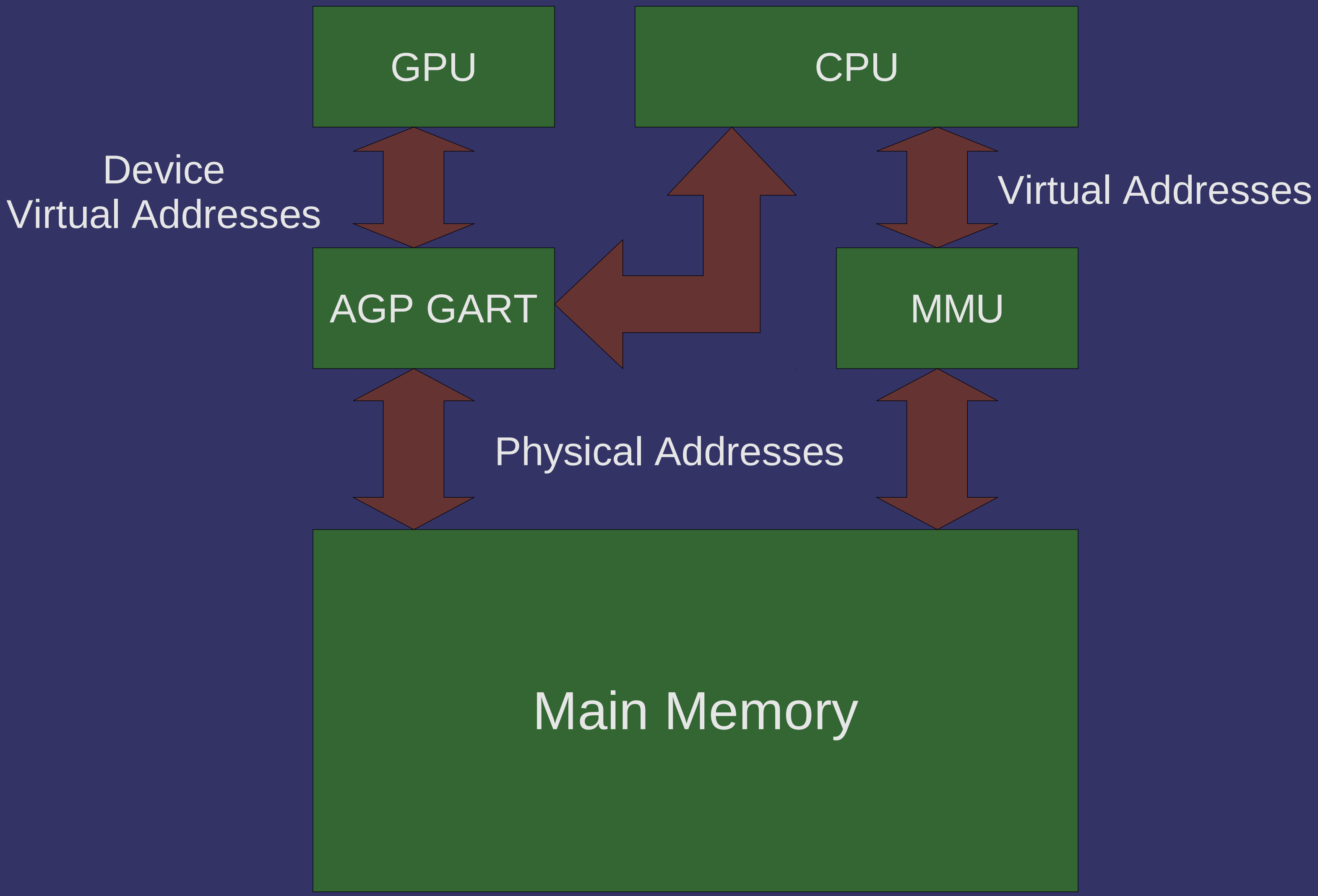



12-October-2011

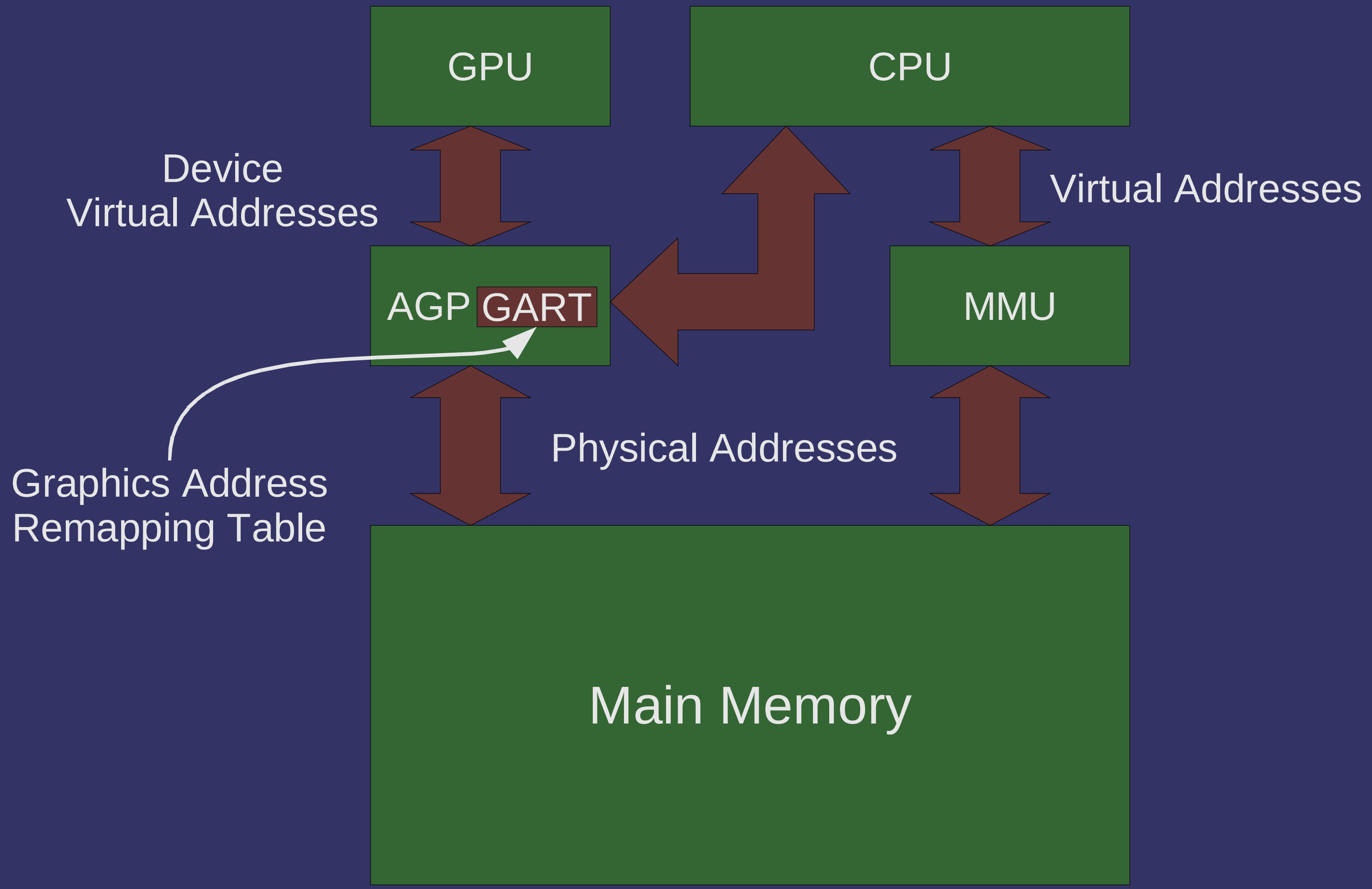
# Unified Memory Architecture



# Memory Map


GPU
CPU
Device
Virtual Addresses
Virtual Addresses
AGP GART
MMU
Physical Addresses
Main Memory

# Memory Map




12-October-2011

# *Vertex Memory*

- Practically, the GPU can only access:
  - Memory physically on the graphics card
  - Memory mapped in the GART
- Only the driver can provide GART or card memory
  - The driver controls what's available to the GPU
  - The driver knows how much memory is available
  - The driver controls the GART mappings
  - The driver knows which memory pool to use
    - ...but we have to give it some hints

# *Vertex Memory*

- In OpenGL this memory is called *buffer object*
  - Used somewhat like a file:
    - Accessed via an opaque handle
    - Bulk I/O “accessor” routines
    - Direct mapping and access via a pointer
      - See http://en.wikipedia.org/wiki/Memory-mapped_file

# Buffer Objects

- Generate “names” for the buffer objects:

  `glGenBuffers(GLsizei num, GLuint *names);`

- “Bind” a buffer for use:

  `glBindBuffer(GLenum target, GLuint name);`

  – `target` selects which buffer we're talking about
    – `GL_ARRAY_BUFFER` is used for vertex data
    – `GL_ELEMENT_ARRAY_BUFFER` is used for vertex indices
      – More on that and other targets later in the term
  – Binding creates the object, but it still has no storage
    – Like creating an empty file

# *Buffer Objects*

- Storage is created and *optionally* initialized with:

```
void glBufferData(GLenum target,
    GLsizeiptr size, const GLvoid *data,
    GLenum usage);
```

  – `usage` tells the GL how the app will utilize the buffer

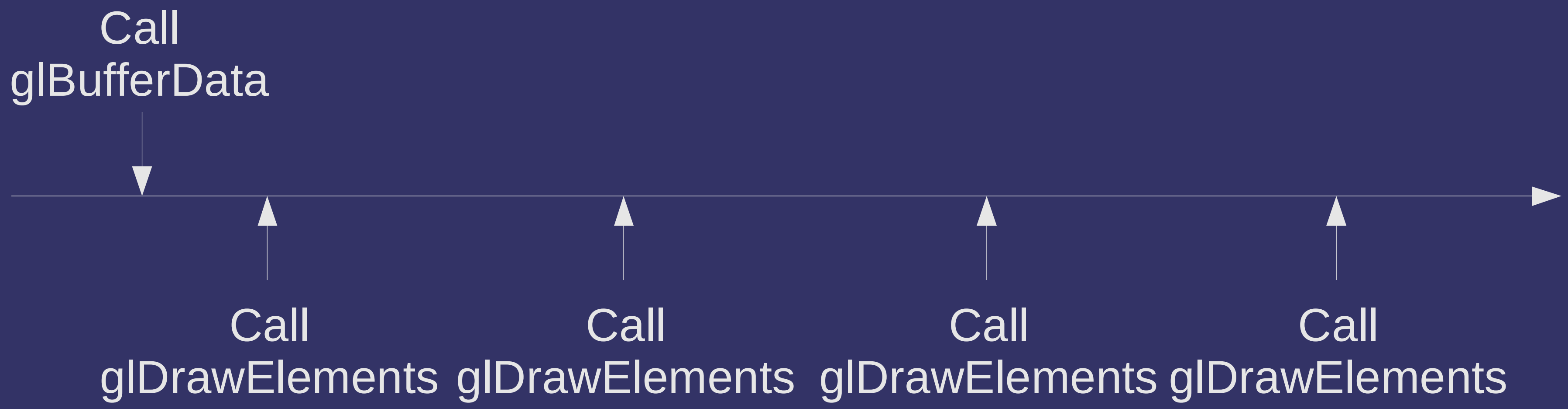
- Storage is updated with:

```
void glBufferSubData(GLenum target,
    GLintptr offset, GLsizeiptr size,
    const GLvoid *data);
```

# *Buffer Object Usage Hints*

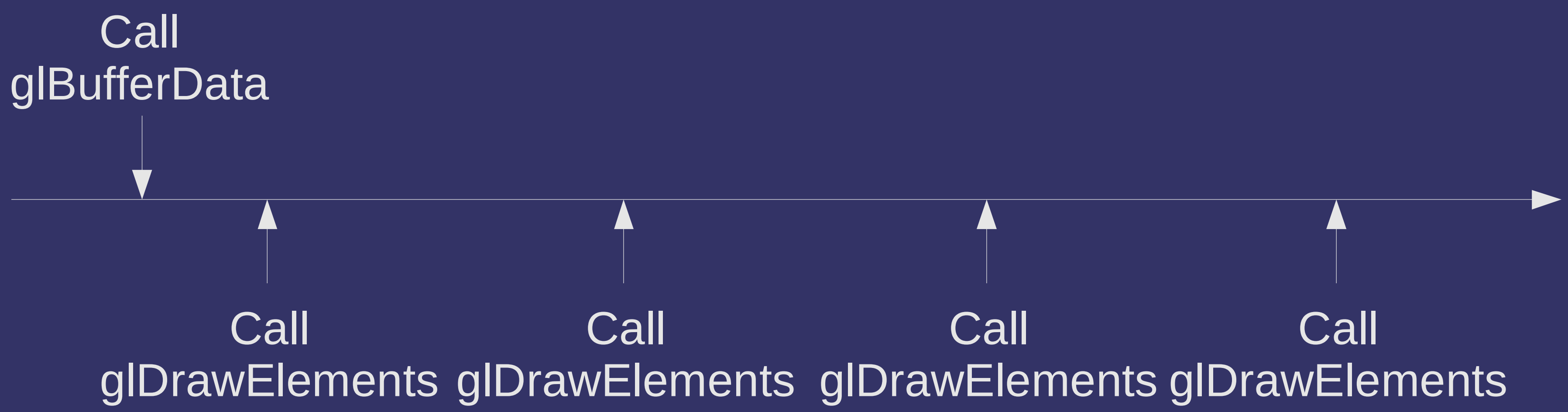
- Usage conveys information along two axes:
  - Data “frequency”:
    - Stream – data is specified once and used a few times
    - Static – data is specified once and used many times
    - Dynamic – data is specified and used many times
  - Data “usage”:
    - Draw – data used as source for drawing
    - Read – data copied from GL and read back to client
    - Copy – data copied from GL and used as source for drawing
  - Combine these to create the enums (e.g., `GL_STATIC_DRAW`)

# *Buffer Object Usage Hints*

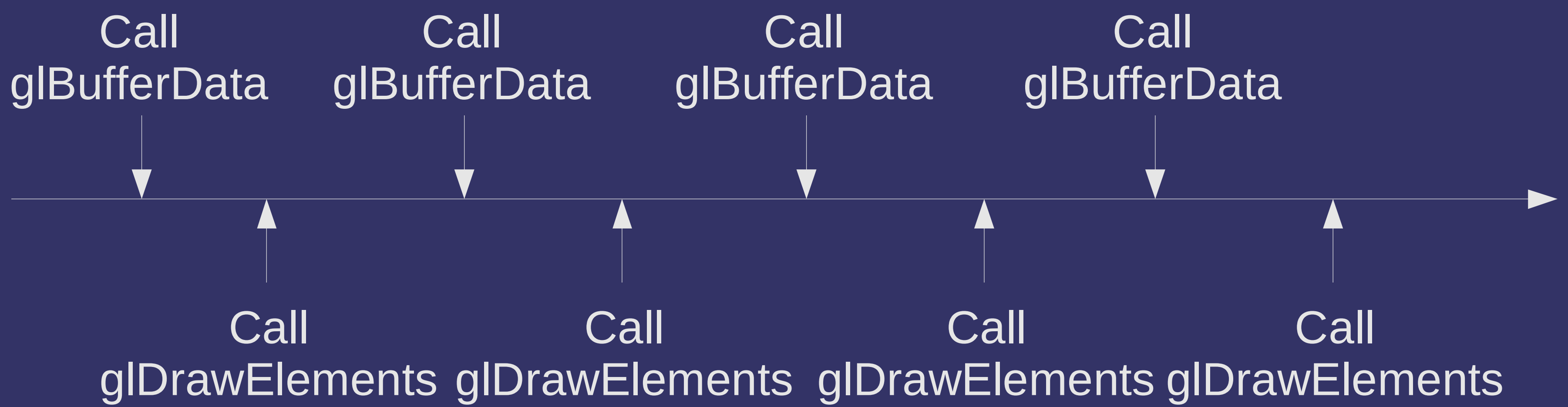


# *Buffer Object Usage Hints*

`GL_STATIC_DRAW`

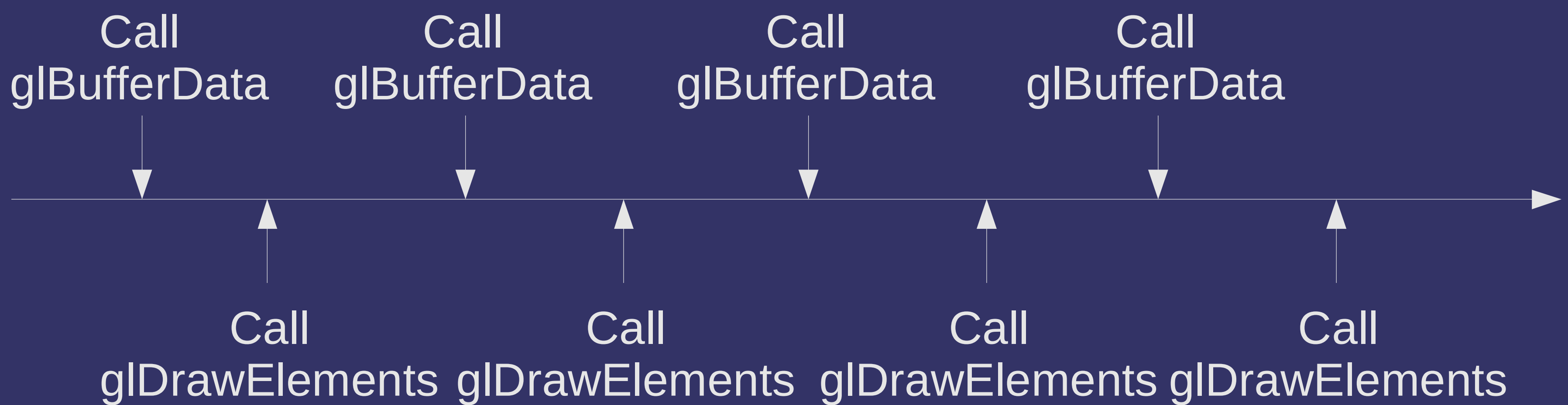


# Buffer Object Usage Hints



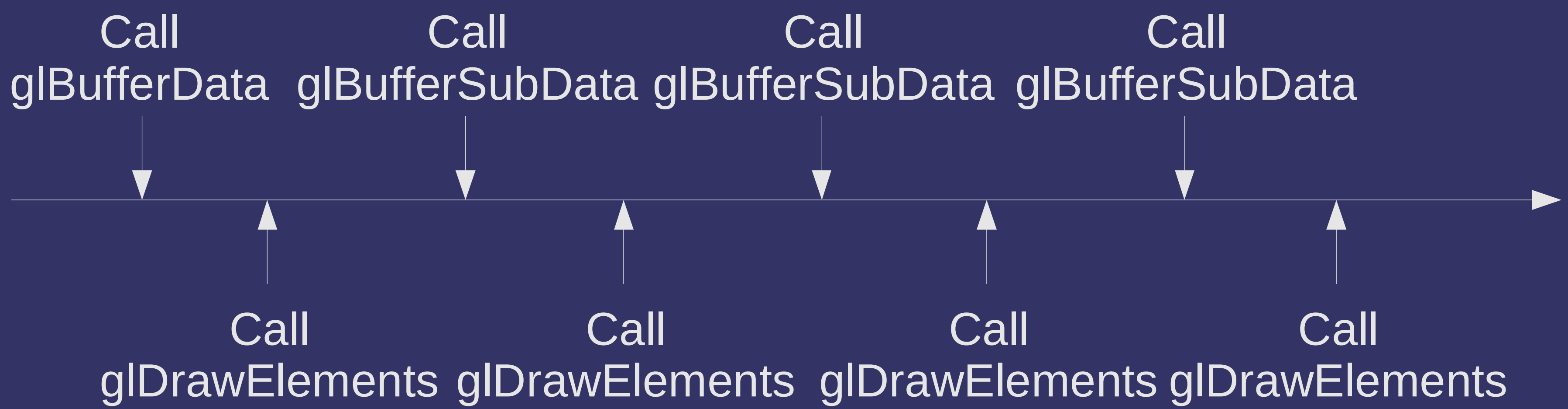
# *Buffer Object Usage Hints*

## `GL_STREAM_DRAW`



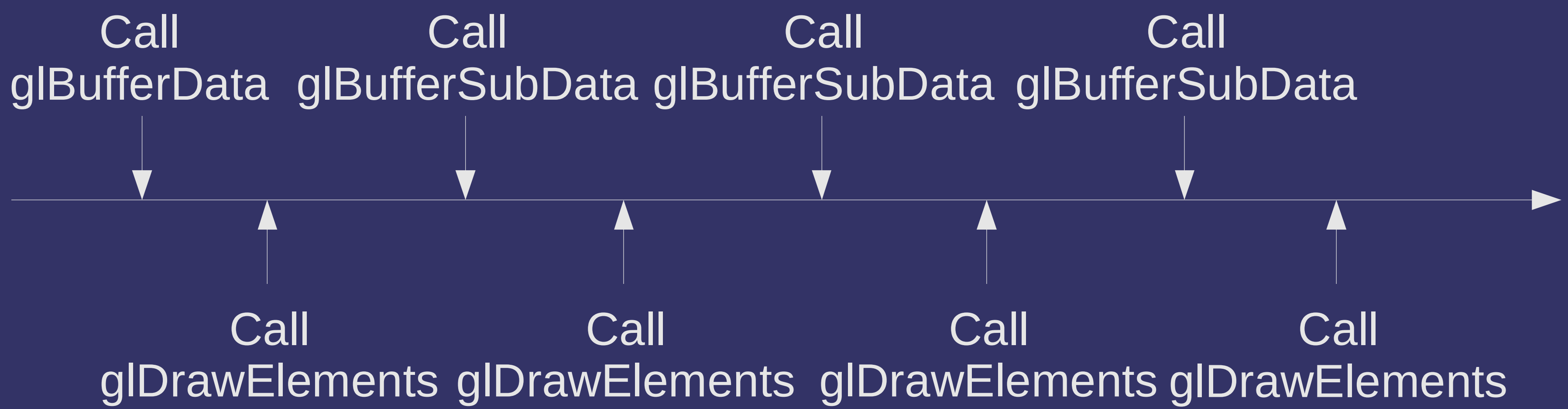

12-October-2011

# Buffer Object Usage Hints



# *Buffer Object Usage Hints*

`GL_DYNAMIC_DRAW`



# Buffer Objects

- Memory backing the buffer can be mapped into CPU space:

```
GLvoid *glMapBuffer(GLenum target,
                    GLenum access);
```

  - `access` tells the driver how the application will access the mapped buffer:
    - `GL_READ_ONLY`
    - `GL_WRITE_ONLY`
    - `GL_READ_WRITE`
- Unmap the buffer with:

```
GLboolean glUnmapBuffer(GLenum target);
```

# *Now what?*

- The vertex data is in a buffer object...how do we tell the GPU know where to get it?

# *Vertex Attribute Pointer*

- Set the location and format of a vertex attribute with:

```
void glVertexAttribPointer(GLuint index,
    GLint size, GLenum type,
    GLboolean normalized, GLsizei stride,
    const GLvoid *pointer);
```

# *Vertex Attribute Pointer*

- Set the location and format of a vertex attribute with:

```
void glVertexAttribPointer(GLuint index,
     GLint size, GLenum type,
     GLboolean normalized, GLsizei stride,
     const GLvoid *pointer);
```

In the API, attributes are numbered

# *Vertex Attribute Pointer*

- Set the location and format of a vertex attribute with:

```
void glVertexAttribPointer(GLuint index,
     GLint size, GLenum type,
     GLboolean normalized, GLsizei stride,
     const GLvoid *pointer);
```

Number of components in each element

Type of data (e.g., `GL_FLOAT`)

# *Vertex Attribute Pointer*

- Set the location and format of a vertex attribute with:

```
void glVertexAttribPointer(GLuint index,
    GLint size, GLenum type,
    GLboolean normalized, GLsizei stride,
    const GLvoid *pointer);
```

For integer data, specifies whether it is normalized or not

# *Vertex Attribute Pointer*

- Set the location and format of a vertex attribute with:

```
void glVertexAttribPointer(GLuint index,
    GLint size, GLenum type,
    GLboolean normalized, GLsizei stride,
    const GLvoid *pointer);
```

Number of bytes from
the start of one element
to the start of the next

# *Vertex Attribute Pointer*

- Set the location and format of a vertex attribute with:

```
void glVertexAttribPointer(GLuint index,
    GLint size, GLenum type,
    GLboolean normalized, GLsizei stride,
    const GLvoid *pointer);
```

Offset, in bytes, from the start of the buffer where the data starts

# *Enable Attribute*

- Attributes that will be used must also be enabled:

```
void glEnableVertexAttribArray(GLuint index);
```

- Attributes can later be disabled:

```
void glDisableVertexAttribArray(GLuint index);
```

# *Setting Attribute Numbers*

- GLSL uses names for attributes:

```
attribute vec4 color;
```

- The API uses numbers:

```
void glVertexAttribPointer(GLuint index,
    GLint size, GLenum type,
    GLboolean normalized, GLsizei stride,
    const GLvoid *pointer);
```

- How do we connect the two?

# Setting Attribute Numbers

- Bind the attribute name to the index we want:

```
void glBindAttribLocation(GLuint programObj,
    GLuint index, const GLchar *name);
```

  - Can only call *before* linking the program
  - Changes to attribute locations do not take effect until the program is linked (or linked again)

# *Drawing*

- Draw a series of vertices:

```
void glDrawArrays(GLenum mode, GLint first,
    GLsizei count);
```

# *Drawing*

- Draw a series of vertices:

```
void glDrawArrays(GLenum mode, GLint first,
    GLsizei count);
```

Sets the primitive type

# Drawing

- Draw a series of vertices:

```
void glDrawArrays(GLenum mode, GLint first,
    GLsizei count);
```

Number of vertices to draw

Selects which vertex in the buffer to start drawing with

# *Primitive Types*

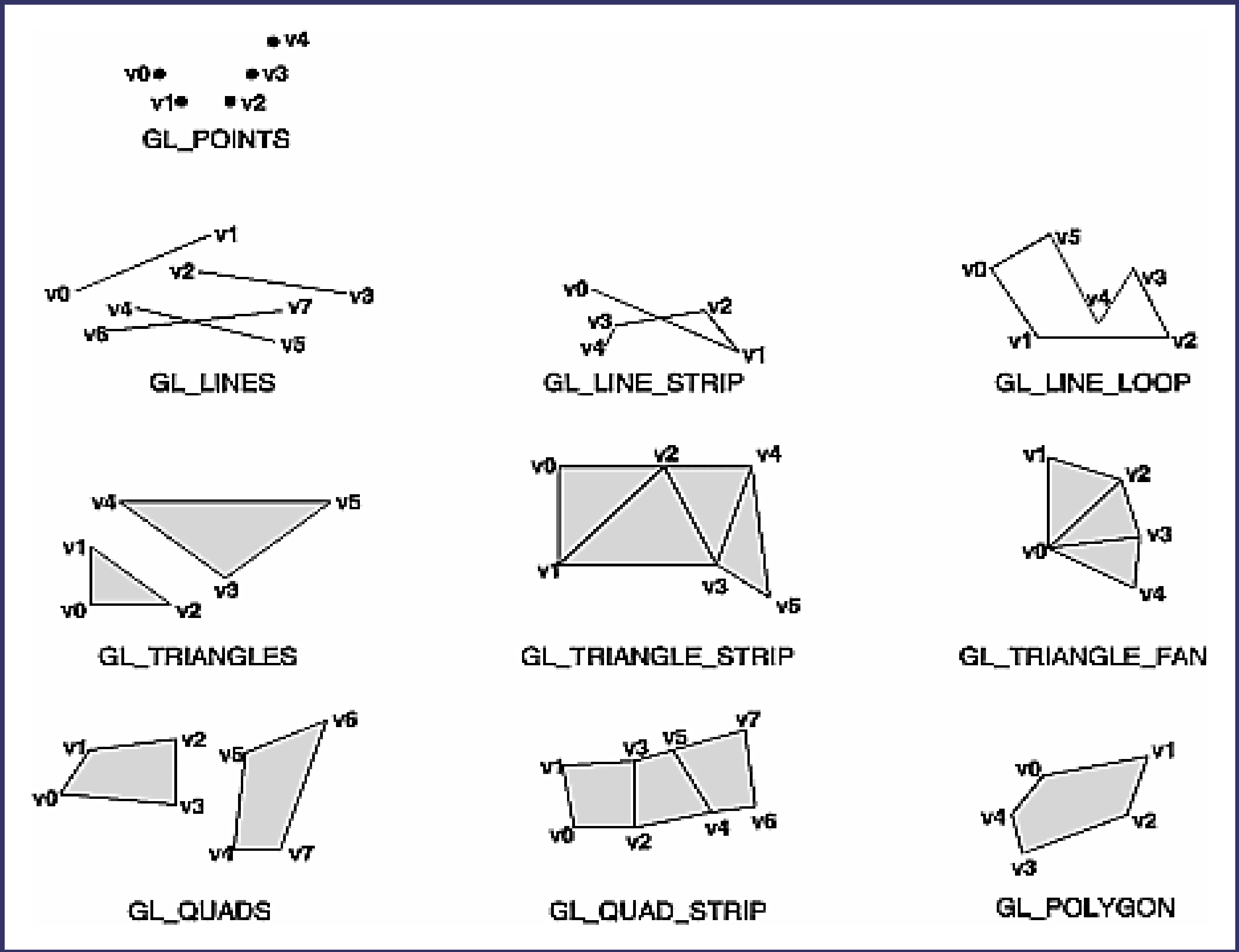



Image borrowed from "OpenGL Programming Guide".

# *Primitive Types*

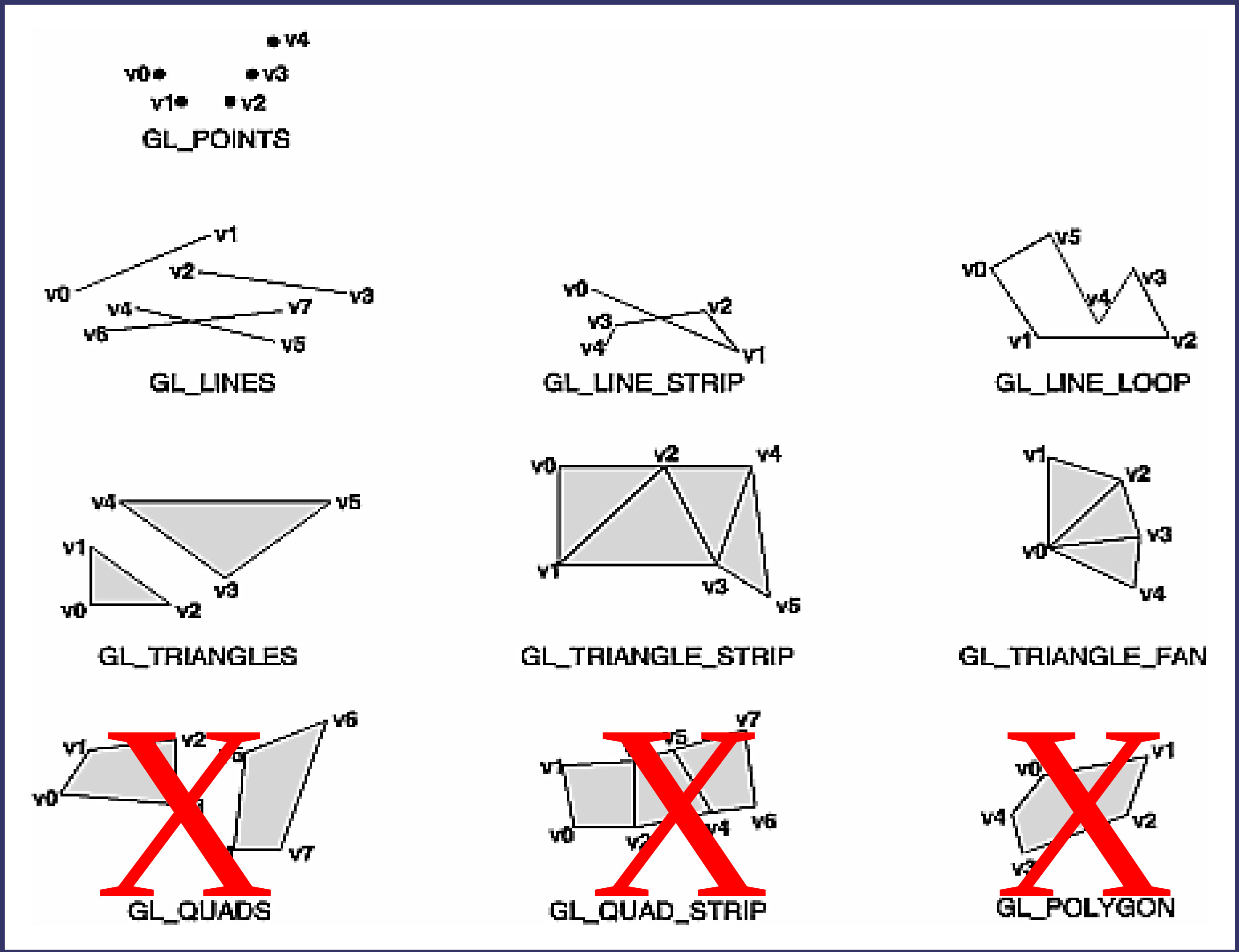



Image borrowed from “OpenGL Programming Guide”.

# *Uniforms*

- Data that changes *at most* per draw call
  - Set using `glUniform` and `glUniformMatrix` commands

# Uniforms

- Single elements can be set:

```
void glUniform4f(GLint location, GLfloat v0,
    GLfloat v1, GLfloat v2, GLfloat v3);
```

  - Also `1f`, `2f`, `3f`, `1i`, `2i`, `3i`, and `4i` variants
  - Number of fields and base type must match the uniform's type
    - `4f` for `vec4`
    - `1i` for `int`
    - etc.

# *Uniforms*

- Multiple array elements can be set:

```
void glUniform4fv(GLint location,
    GLsizei count, const GLint *value);
```

  - Also `1fv`, `2fv`, `3fv`, `1iv`, `2iv`, `3iv`, and `4iv` variants
  - Number of fields and base type must match the uniform's type
  - `count` is the number of array elements to set

# *Uniforms*

- Matrices can be set:

```
void glUniformMatrix4fv(GLint location,
    GLsizei count, GLboolean transpose,
    const GLfloat *value)
```

  - Many variants for different matrix sizes
    - Size must match the dimension of the uniform
  - `count` is the number of matrices to set
  - `transpose` specifies whether the supplied data is column-major or row-major
    - The OpenGL default is **always** column-major

# *Uniform Locations*

- What is this “location”?

```
void glUniform4fv(GLint location,
    GLsizei count, const GLfloat *value)
```

  - Like the index for attributes, but set by the linker in-stead of the applicatoin
  - The linker assigns a slot for each *active* uniform

# *Uniform Locations*

- Query the location

```
GLint glGetUniformLocation(GLuint program,
    const GLchar *name);
```

  - `program` is a linked program object
  - `name` is the uniform to locate
    - Can be a scalar, vector, or matrix
    - Can be a whole array or an array element
    - Can be a field of a structure
    - Cannot be a whole structure.
  - Returns -1 if the requested uniform does not exist or is not active

# Uniform Locations

```
// vertex shader
uniform vec4 v;
uniform mat4 m;
attribute vec4 a;

void main() {
    gl_Position = m * a;
}

// fragment shader
uniform vec3 c;

void main() {
    gl_FragData[0] =
        vec4(c, 1.0);
}
```

# *Uniform Locations*

```
// vertex shader
uniform vec4 v;
uniform mat4 m;
attribute vec4 a;

void main() {
    gl_Position = m * a;
}

// fragment shader
uniform vec3 c;

void main() {
    gl_FragData[0] =
        vec4(c, 1.0);
}
```

These uniforms are active.

# *Uniform Locations*

```
// vertex shader
uniform vec4 v;
uniform mat4 m;
attribute vec4 a;

void main() {
    gl_Position = m * a;
}

// fragment shader
uniform vec3 c;

void main() {
    gl_FragData[0] =
        vec4(c, 1.0);
}
```

This uniform is not.

These uniforms are active.

# References

- More information about I/O MMUs in general:

  http://en.wikipedia.org/wiki/IOMMU

- Nvidia whitepaper about using VBOs:

  http://developer.nvidia.com/object/using_VBOs.html

# *Linear Algebra Primer*

- Three important data types:
  - Scalar values
  - Row / column vectors
    - 1x4 and 4x1 are the most common sizes
  - Square matrices
    - 4x4 is the most common size...to match the 1x4 & 4x1 vectors

# Notation

- Try to use the same notation as the textbook:
  - Angle: $\theta$ (lower-case Greek)
  - Scalar: $s$ (lower-case, italic, serif)
  - Vector or point: $\mathbf{v}$ (lower-case, bold, serif)
    - Sometimes $\hat{\mathbf{u}}$ is used to differentiate vectors from points
  - Matrix: $\mathbf{M}$ (upper-case, bold, serif)
  - Plane: $\pi$: $\mathbf{n} \cdot x + d$ ($\pi$: a vector and a scalar)
  - Triangle: $\triangle\mathbf{abc}$ ($\triangle$ 3 points)
  - Line segment: $\mathbf{ab}$ (2 points)
  - Geometric entity: $A$ (upper-case, italic, serif)

# *Row Vectors*

- These are special matrices that have multiple columns but only one row
  - Example: $\begin{bmatrix} 5.0 & 3.14 & 37 \end{bmatrix}$
- Addition and subtraction is component-wise:
  - Example: $\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 9 & 10 & 11 \end{bmatrix} = \begin{bmatrix} 10 & 12 & 14 \end{bmatrix}$
  - Both vectors must be the same size
- Operations with scalars also component-wise:
  - Example: $3.2 \times \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 3.2 & 6.4 & 9.6 \end{bmatrix}$
- Notice that vector multiplication is missing...

# Column Vectors

- These are special matrices that have multiple rows but only one column
  - Example: $\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$
- Work just like row vectors
- Notationally convert a row to a column with a $\mathrm{T}$ in the exponent
  - Example: $\mathbf{v}^T$
  - We'll talk more about this notation later...

# *Vector Operations*

- There are a few operations specific to vectors that are really important to graphics:
  - Dot product
  - Vector magnitude / normalization
  - Cross product

# Dot Product

- Component-wise multiply, then sum components
  - Example:

$$\begin{bmatrix} 2.3 & 1.2 \end{bmatrix} \cdot \begin{bmatrix} 1.7 & 6.5 \end{bmatrix} = (2.3 * 1.7) + (1.2 * 6.5) = 11.71$$

  - Noted as $\mathbf{u} \cdot \mathbf{v}$ or $\langle \mathbf{u}, \mathbf{v} \rangle$
  - Also known as the *inner product* or *scalar product*

# *Vector Magnitude*

- Noted by vertical bars around the vector
  - Like absolute value...which is the scalar magnitude
  - Can also be thought of as the length of the vector
- Square-root of dot-product of vector with itself
  - Like absolute value
  - Example: $\left\| \begin{bmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix} \right\| = \sqrt{\begin{bmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix} \cdot \begin{bmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix}} =$

$$\sqrt{\left(\frac{\sqrt{2}}{2}\right)^2 + \left(\frac{\sqrt{2}}{2}\right)^2} = \sqrt{\frac{2}{4} + \frac{2}{4}} = 1$$

# *Normal*

- *Normal* is an overloaded term in graphics and linear algebra
  - Sometimes it means a vector has unit length
    - $|\mathbf{u}| = 1.0$
    - Can say the vector is “normalized”
  - Sometimes it means a vector is perpendicular to a surface or another vector
    - This mean the angle between the vectors is 90°
    - Can say that the vectors are “normal to each other”
    - Can say that the vectors are “orthogonal”
  - Can combine for even more fun!
    - “Use normalized surface normals in the calculation.”

# Normalize

- Can normalize a vector by dividing it by its magnitude
  - Example: $\dfrac{\mathbf{u}}{|\mathbf{u}|}$
  - Vector has the same direction, but the magnitude will be 1.0
  - Also works with scalars

# *Dot Product*

- Why is the dot product so interesting?

# *Dot Product*

- Why is the dot product so interesting?
  - The dot product of two vectors is related to the cosine of the angle between those vectors
  - Formally: $\mathbf{u} \cdot \mathbf{v} = |\mathbf{u}|\ |\mathbf{v}| \cos\theta$
- We often want to know the angle between two vectors
  - This is the basis of all lighting calculations in 3D graphics!

$$\frac{\mathbf{u} \cdot \mathbf{v}}{|\mathbf{u}||\mathbf{v}|} = \frac{\mathbf{u}}{|\mathbf{u}|} \cdot \frac{\mathbf{v}}{|\mathbf{v}|} = \cos\theta$$

# *Cross Product*

- From Wikipedia:

  [T]he cross product is a binary operation on two vectors in a three-dimensional Euclidean space that results in another vector which is perpendicular to the plane containing the two input vectors.

  - Noted as an $\times$ between two vectors
  - Calculated as:

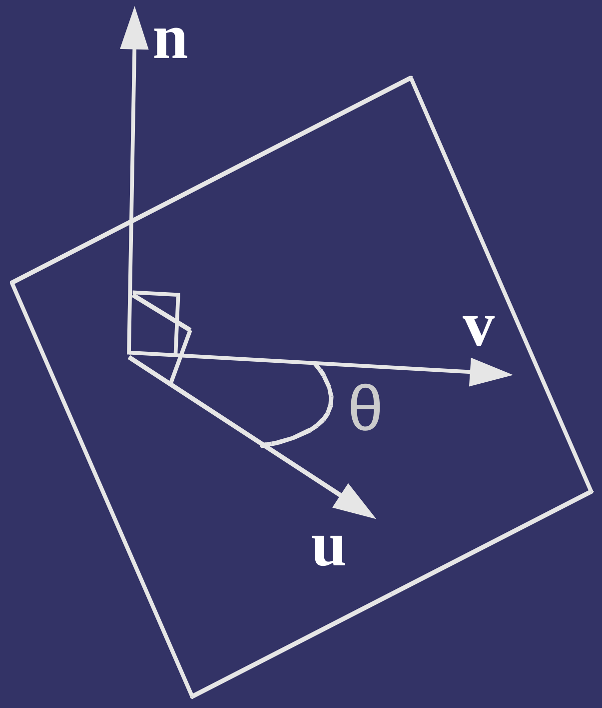
$$\mathbf{a}\times\mathbf{b}=\left[\mathbf{a}_y\mathbf{b}_z-\mathbf{a}_z\mathbf{b}_y \quad \mathbf{a}_z\mathbf{b}_x-\mathbf{a}_x\mathbf{b}_z \quad \mathbf{a}_x\mathbf{b}_y-\mathbf{a}_y\mathbf{b}_x\right]$$

  - Not associative
  - Anti-commutative: If $\mathbf{u}\times\mathbf{v} = \mathbf{w}$, then $\mathbf{v}\times\mathbf{u} = -\mathbf{w}$

[1] From http://en.wikipedia.org/wiki/Cross_product

# *Cross Product*

- Why is the cross product so interesting?
  - Cross product of two vectors results in a new vector that is normal both
  - The cross product of two vectors is related to the sine of the angle between the vectors
    - Formally: $\mathbf{u} \times \mathbf{v} = |\mathbf{u}|\ |\mathbf{v}| \sin\theta\ \mathbf{n}$



# *Matrices*

- Like vectors, but have multiple rows and columns
  - Example: $\begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$
- Add and subtract like you would expect
  - Like vectors, both matrices must be the same size...in both dimensions

# Matrix Multiplication

- Special rules make matrix multiplication different from scalar multiplication
  - **NOT** commutative!  e.g., $\mathbf{M}\times\mathbf{N} \neq \mathbf{N}\times\mathbf{M}$
  - Associative  e.g., $\mathbf{A}(\mathbf{BC}) = (\mathbf{AB})\mathbf{C}$
  - Column count of first matrix must match row count of second matrix
    - If $\mathbf{M}$ is 4-by-3 matrix and $\mathbf{N}$ is a 3-by-1 matrix, we can do $\mathbf{M}\times\mathbf{N}$ but not $\mathbf{N}\times\mathbf{M}$
  - If the source matrices are $n$-by-$m$ and $m$-by-$p$, the re-sulting matrix will be $n$-by-$p$

# Matrix Multiplication

- To calculate an element of the matrix, **C**, resulting from **AB**:

$$\mathbf{C}_{ij} = \Sigma_{r=1}^{n} \mathbf{A}_{ir} \mathbf{B}_{rj}$$
$$= \mathbf{A}_{i,0} \mathbf{B}_{0,j} + \mathbf{A}_{i,1} \mathbf{B}_{1,j} + \mathbf{A}_{i,2} \mathbf{B}_{2,j} + \ldots + \mathbf{A}_{i,n} \mathbf{B}_{n,j}$$

- What does this look like?

# Matrix Multiplication

- To calculate an element of the matrix, $\mathbf{C}$, resulting from $\mathbf{AB}$:

$$\begin{aligned}\mathbf{C}_{ij} &= \Sigma_{r=1}^{n} \mathbf{A}_{ir} \mathbf{B}_{rj} \\ &= \mathbf{A}_{i,0}\mathbf{B}_{0,j} + \mathbf{A}_{i,1}\mathbf{B}_{1,j} + \mathbf{A}_{i,2}\mathbf{B}_{2,j} + \ldots + \mathbf{A}_{i,n}\mathbf{B}_{n,j}\end{aligned}$$

- What does this look like?
  - The dot product of a row of $\mathbf{A}$ with a column of $\mathbf{B}$!
  - This is why the column count of $\mathbf{A}$ must match the row count of $\mathbf{B}$...otherwise the dot product wouldn't work

# *Multiplicative Identity*

- There is a multiplicative identity for matrices

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

  - Just like any other multiplicative identity, $\mathbf{AI} = \mathbf{A}$
  - If you pretend that a scalar is a $1 \times 1$ matrix, this should make sense

# *Transpose*

- Rows become columns and columns become rows
  - Noted with a T in the exponent position (e.g., $\mathbf{M}^T$)
  - Example:

$$\begin{bmatrix} 2 & 3 \\ 4 & 5 \\ 6 & 7 \end{bmatrix}^T = \begin{bmatrix} 2 & 4 & 6 \\ 3 & 5 & 7 \end{bmatrix}$$

# *Matrix Multiplication*

- Can rewrite the dot product (inner product) of two row vectors as:

$$s = \mathbf{u}\,\mathbf{v}^{\mathrm{T}}$$

- Can write the *outer product* of two row vectors as:

$$\mathbf{M} = \mathbf{u}^{\mathrm{T}}\,\mathbf{v}$$

  - Notation is $\mathbf{u} \otimes \mathbf{v}$

$$\mathbf{u} \otimes \mathbf{v} = \begin{bmatrix} \mathbf{u}_1\mathbf{v}_1 & \mathbf{u}_1\mathbf{v}_2 & \mathbf{u}_1\mathbf{v}_3 & \ldots & \mathbf{u}_1\mathbf{v}_n \\ \mathbf{u}_2\mathbf{v}_1 & \mathbf{u}_2\mathbf{v}_2 & \mathbf{u}_2\mathbf{v}_3 & \ldots & \mathbf{u}_2\mathbf{v}_n \\ \cdots & \cdots & \ddots & & \cdots \\ \mathbf{u}_m\mathbf{v}_1 & \mathbf{u}_m\mathbf{v}_2 & \mathbf{u}_m\mathbf{v}_3 & \ldots & \mathbf{u}_m\mathbf{v}_n \end{bmatrix}$$

# *Matrix Multiplication*

- Not commutative

$$\mathbf{M} \times \mathbf{N} \neq \mathbf{N} \times \mathbf{M}$$

- But...

$$\mathbf{M} \times \mathbf{N} = \left(\mathbf{N}^T \times \mathbf{M}^T\right)^T$$

- How is this useful?

# *Matrix Multiplication*

- Not commutative

$$\mathbf{M} \times \mathbf{N} \neq \mathbf{N} \times \mathbf{M}$$

- But...

$$\mathbf{M} \times \mathbf{N} = \left(\mathbf{N}^T \times \mathbf{M}^T\right)^T$$

- How is this useful?
  - Assume $v$ is a vector we want to transform by a matrix $M$, but we only have $M^T$ in our program...

$$\mathbf{M} \times \mathbf{v} = \left(\mathbf{v}^T \times \mathbf{M}^T\right)^T$$

  - A vector and its transpose are represented the same way (`vec4` in GLSL), so we don't have to do the transpose of the matrix

# *References*

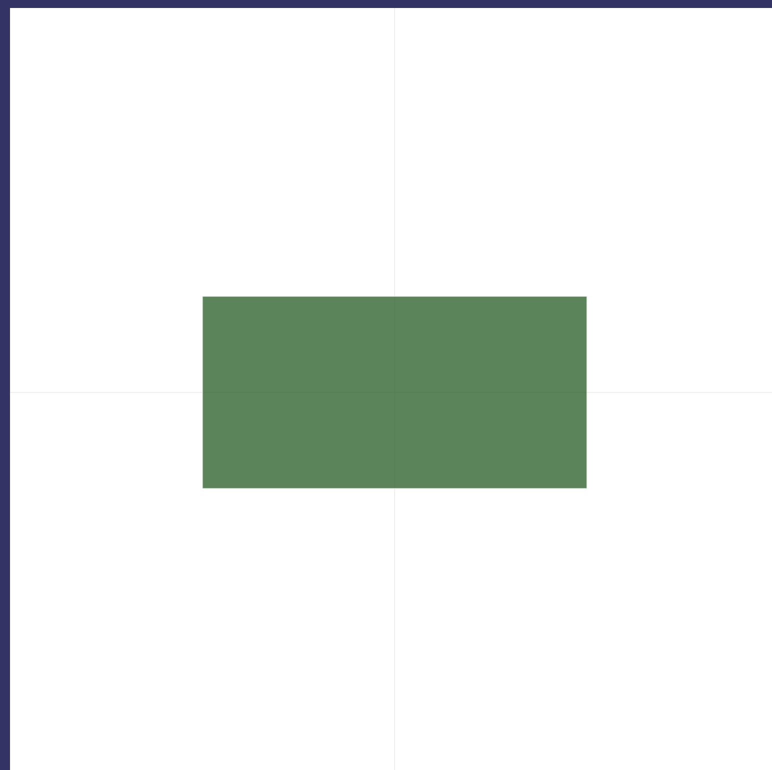
http://en.wikipedia.org/wiki/Matrix_multiplication

http://en.wikipedia.org/wiki/Dot_product

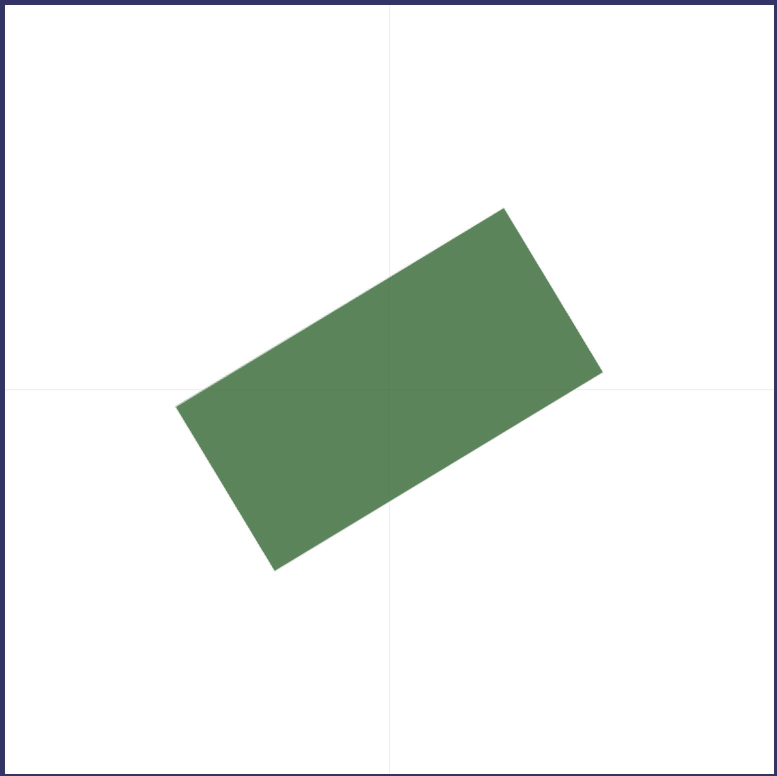
http://en.wikipedia.org/wiki/Cross_product

http://en.wikipedia.org/wiki/Outer_product

# *Rotation*

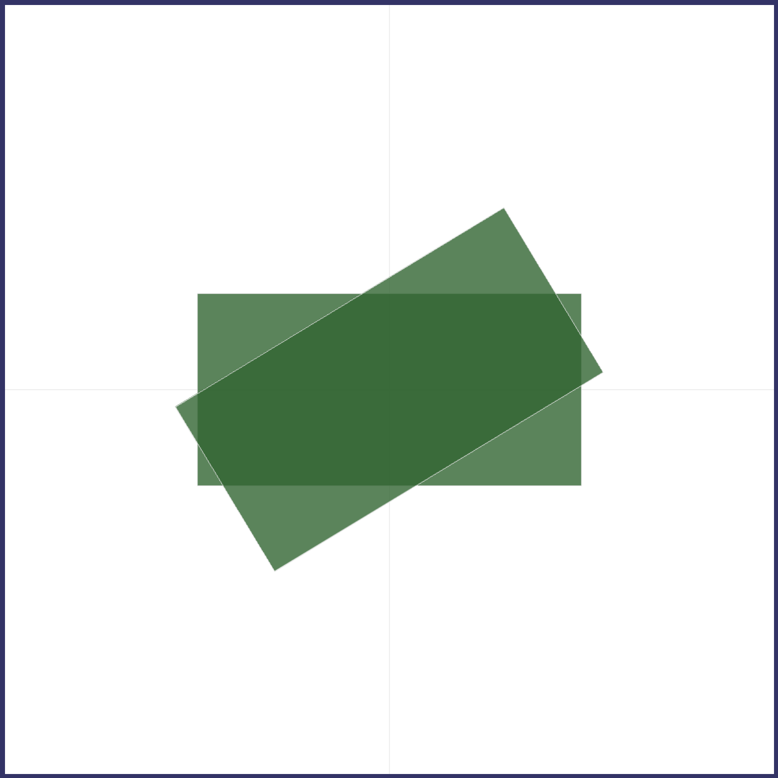


# *Rotation*

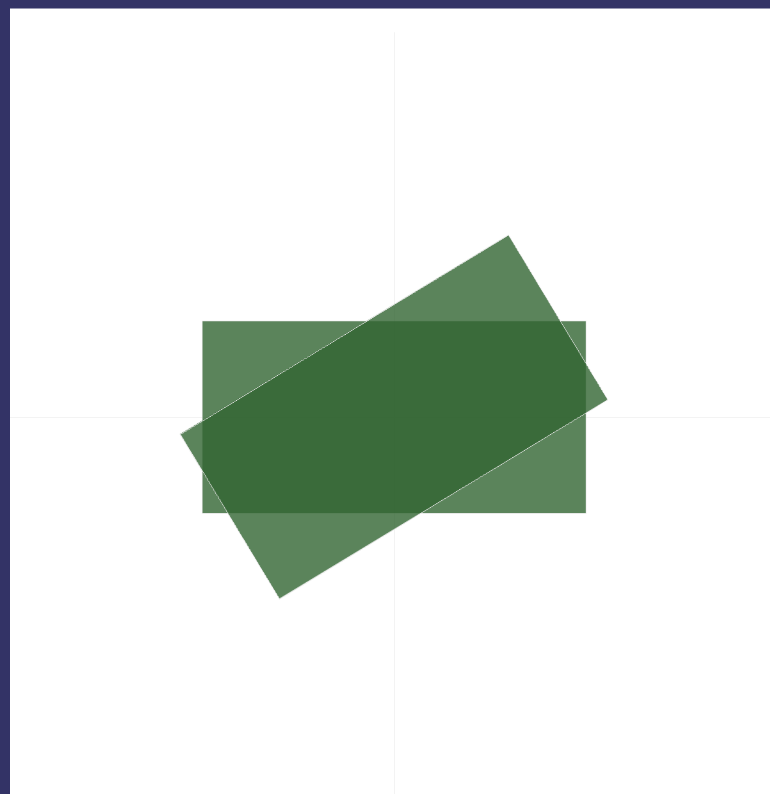


12-October-2011

# *Rotation*



# *Rotation*



$$\mathbf{x}\cos\theta - \mathbf{y}\sin\theta$$
$$\mathbf{x}\sin\theta + \mathbf{y}\cos\theta$$

12-October-2011

# Rotation

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

# Rotation

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

$$\Downarrow$$

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

# *Rotation*

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

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

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

# *Rotation*

- Rotation around the Z-axis
  - If $\theta$ is $0^\circ$, this is the identity matrix

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Rotation around the Y-axis

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

# Rotation

- Why use the matrix method?
  - We can rotate using 4 multiplies and 2 adds
  - A matrix multiply requires 16 multiplies and 12 adds

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

# Rotation

- A series of rotations can be implemented as:

$$\begin{aligned}\mathbf{v}' &= \mathbf{M}_1\mathbf{v}\\ \mathbf{v}'' &= \mathbf{M}_2\mathbf{v}'\\ \mathbf{v}''' &= \mathbf{M}_3\mathbf{v}''\end{aligned}$$

- With substitution:

$$\begin{aligned}\mathbf{v}''' &= \mathbf{M}_3\mathbf{v}''\\ &= \mathbf{M}_3\left(\mathbf{M}_2\mathbf{v}'\right)\\ &= \mathbf{M}_3\left(\mathbf{M}_2\left(\mathbf{M}_1\mathbf{v}\right)\right)\end{aligned}$$

  - Why is this useful?

# *Rotation*

- A series of rotations can be implemented as:

$$\begin{aligned} \mathbf{v}' &= \mathbf{M}_1 \mathbf{v} \\ \mathbf{v}'' &= \mathbf{M}_2 \mathbf{v}' \\ \mathbf{v}''' &= \mathbf{M}_3 \mathbf{v}'' \end{aligned}$$

- With substitution:

$$\begin{aligned} \mathbf{v}''' &= \mathbf{M}_3 \mathbf{v}'' \\ &= \mathbf{M}_3 \left( \mathbf{M}_2 \mathbf{v}' \right) \\ &= \mathbf{M}_3 \left( \mathbf{M}_2 \left( \mathbf{M}_1 \mathbf{v} \right) \right) \\ &= \left( \mathbf{M}_3 \mathbf{M}_2 \mathbf{M}_1 \right) \mathbf{v} \end{aligned}$$

  - Matrix multiplication is associative!

# *Rotation*

- A series of rotations can be implemented as:

$$\begin{aligned} \mathbf{v}' &= \mathbf{M}_1 \mathbf{v} \\ \mathbf{v}'' &= \mathbf{M}_2 \mathbf{v}' \\ \mathbf{v}''' &= \mathbf{M}_3 \mathbf{v}'' \end{aligned}$$

The matrices are composed in the <u>reverse</u> order of how they are applied to the vector!

- With substitution:

$$\begin{aligned} \mathbf{v}''' &= \mathbf{M}_3 \mathbf{v}'' \\ &= \mathbf{M}_3 \left( \mathbf{M}_2 \mathbf{v}' \right) \\ &= \mathbf{M}_3 \left( \mathbf{M}_2 \left( \mathbf{M}_1 \mathbf{v} \right) \right) \\ &= \left( \mathbf{M}_3 \mathbf{M}_2 \mathbf{M}_1 \right) \mathbf{v} \end{aligned}$$

  - Matrix multiplication is associative!

# *Arbitrary Rotation*

- Given a vector, $\mathbf{v}$, and an angle, $\theta$, we can create an arbitrary rotation matrix:

$$\tilde{\mathbf{V}} = \begin{bmatrix} 0 & -\mathbf{v}_z & \mathbf{v}_y & 0 \\ \mathbf{v}_z & 0 & -\mathbf{v}_x & 0 \\ -\mathbf{v}_y & \mathbf{v}_x & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R} = \left(\mathbf{I}\cos\theta\right) - \left(\left(1-\cos\theta\right)\left(\mathbf{v}\otimes\mathbf{v}\right)\right) + \left(\tilde{\mathbf{V}}\sin\theta\right)$$

# Translation

- Points are stored as $\mathbf{p} = [\ x\ y\ z\ 1\ ]$
- Remember the definition of matrix multiplication:

$$
\begin{aligned}
\mathbf{p}_x' &= \mathbf{p}_x\mathbf{M}_{11} + \mathbf{p}_y\mathbf{M}_{12} + \mathbf{p}_z\mathbf{M}_{13} + \mathbf{p}_w\mathbf{M}_{14} \\
\mathbf{p}_y' &= \mathbf{p}_x\mathbf{M}_{21} + \mathbf{p}_y\mathbf{M}_{22} + \mathbf{p}_z\mathbf{M}_{23} + \mathbf{p}_w\mathbf{M}_{24} \\
\mathbf{p}_z' &= \mathbf{p}_x\mathbf{M}_{31} + \mathbf{p}_y\mathbf{M}_{32} + \mathbf{p}_z\mathbf{M}_{33} + \mathbf{p}_w\mathbf{M}_{34} \\
\mathbf{p}_w' &= \mathbf{p}_x\mathbf{M}_{41} + \mathbf{p}_y\mathbf{M}_{42} + \mathbf{p}_z\mathbf{M}_{43} + \mathbf{p}_w\mathbf{M}_{44}
\end{aligned}
$$

- Since $\mathbf{p}_w$ is always 1, the 4th column of the matrix acts as a translation

# *Scaling*

- To scale a vector, multiply each component by a scale factor

$$\mathbf{M} = \begin{bmatrix} \mathbf{s}_x & 0 & 0 & 0 \\ 0 & \mathbf{s}_y & 0 & 0 \\ 0 & 0 & \mathbf{s}_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

# *Next week...*

- Quiz #1
  - Will cover material from last week and this week
- More transformations
- Hidden surface removal / occlusion
  - Backface culling
  - Painters algorithm
  - Z-buffer
  - Frustum culling
- Assignment #2, part 1

# *Legal Statement*

This work represents the view of the authors and does not necessarily represent the view of Intel or the Art Institute of Portland.

OpenGL is a trademark of Silicon Graphics, Inc. in the United States, other countries, or both.

Khronos and OpenGL ES are trademarks of the Khronos Group.

Other company, product, and service names may be trademarks or service marks of others.