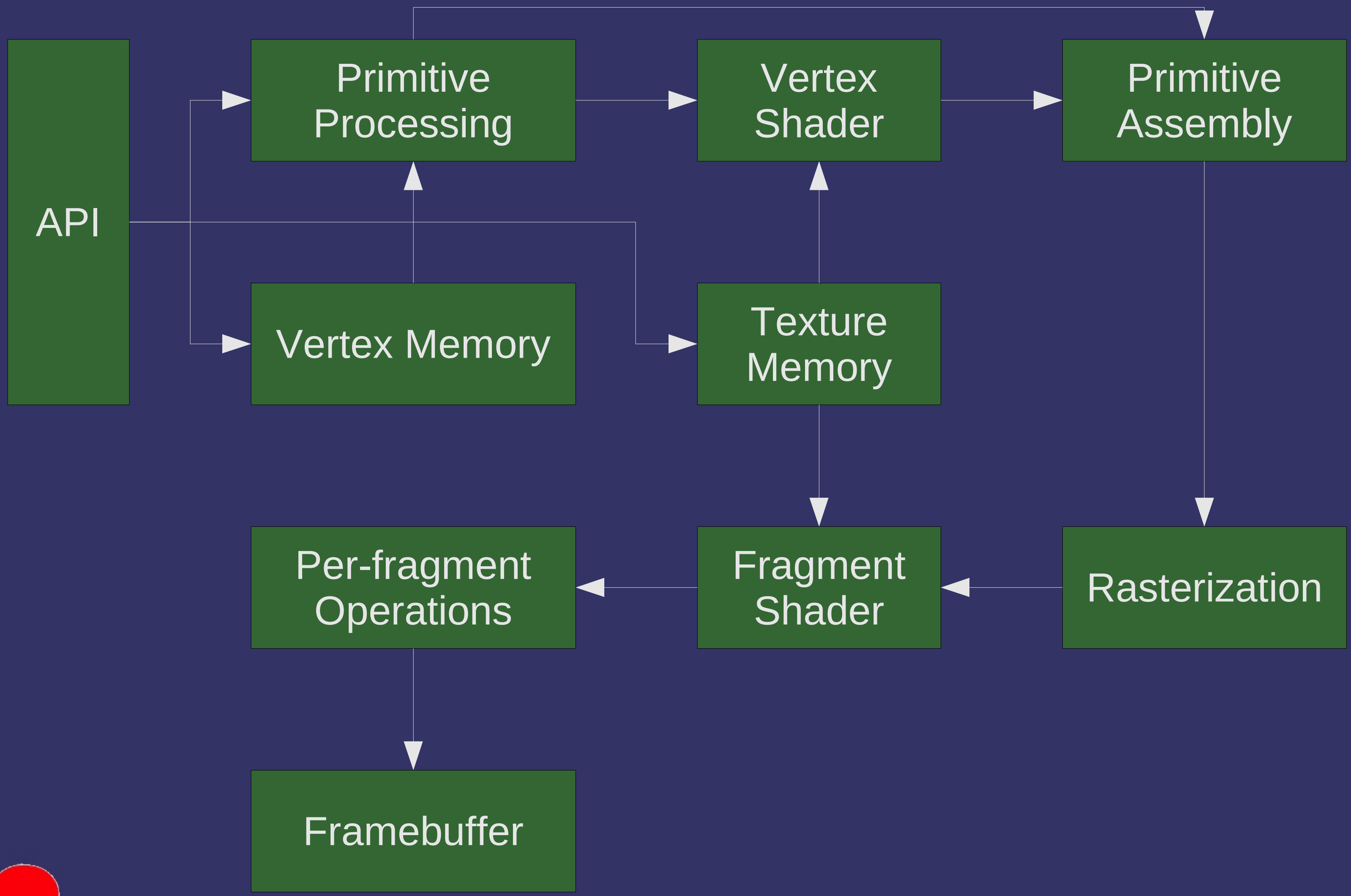
VGP351 – Week 2

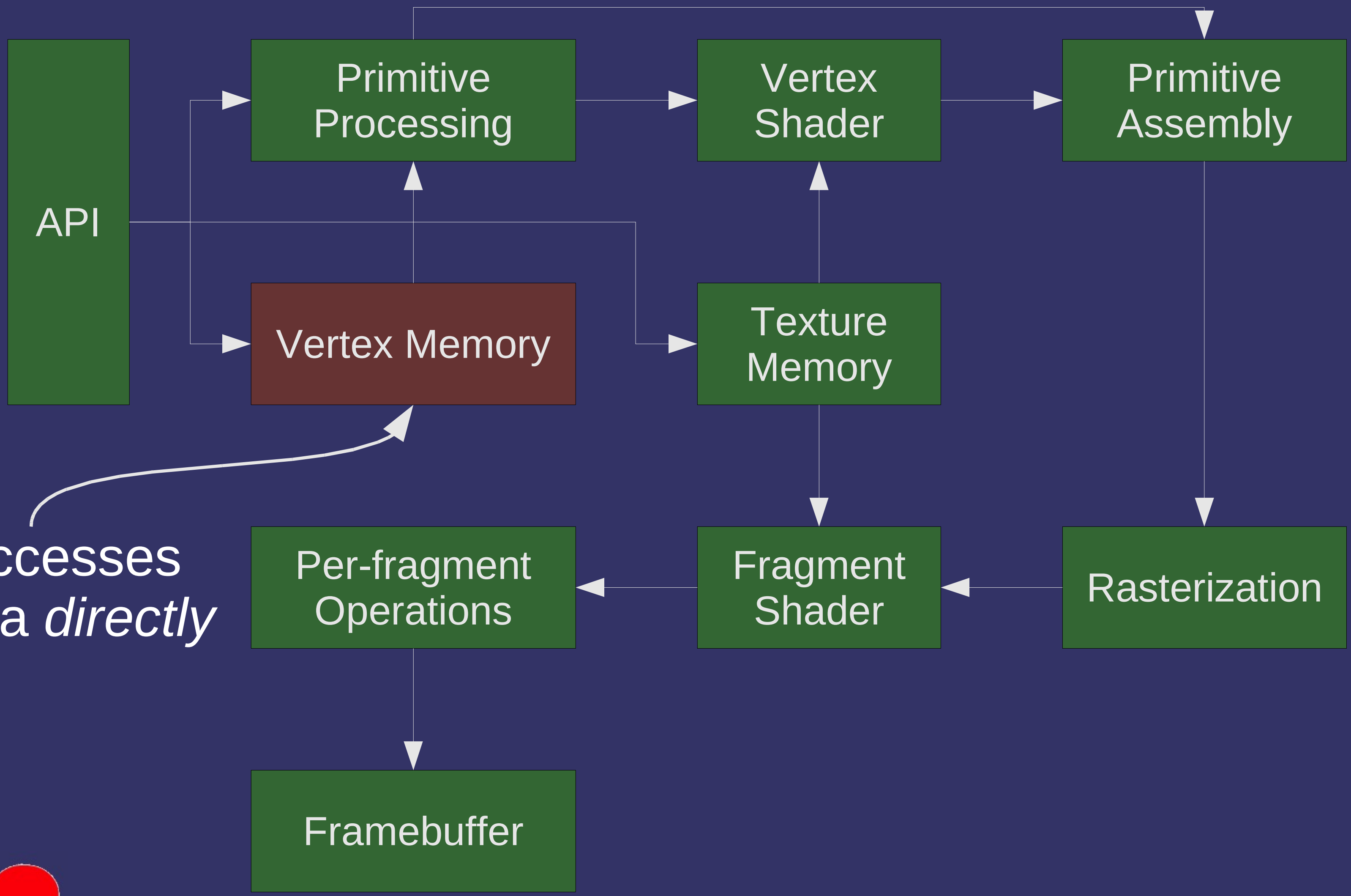
- Agenda:
  - Getting data to the GPU
  - Types of primitives
  - Transformations
    - Modeling
    - Viewing
    - Projection

22-January-2009

Graphics Pipeline

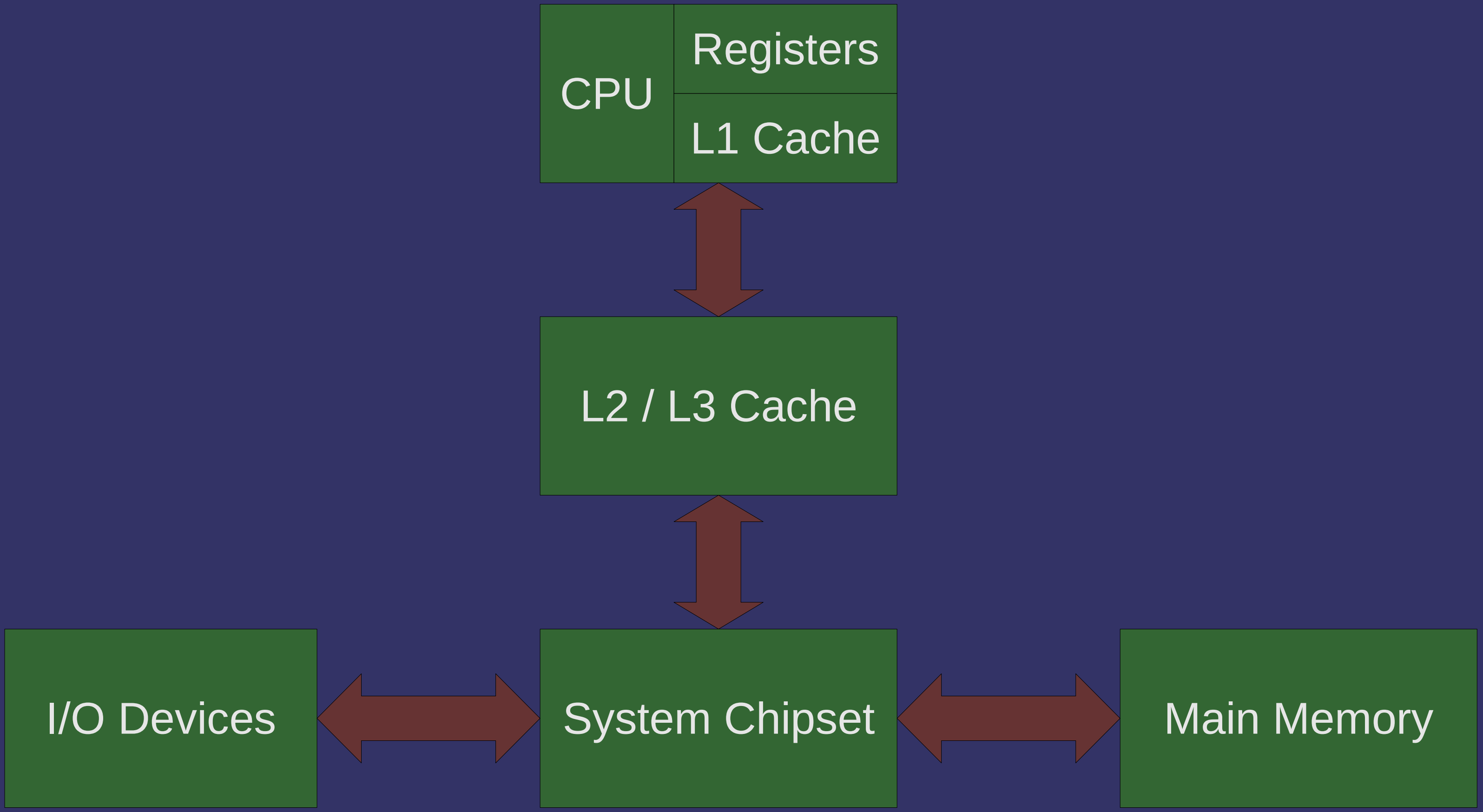


Graphics Pipeline



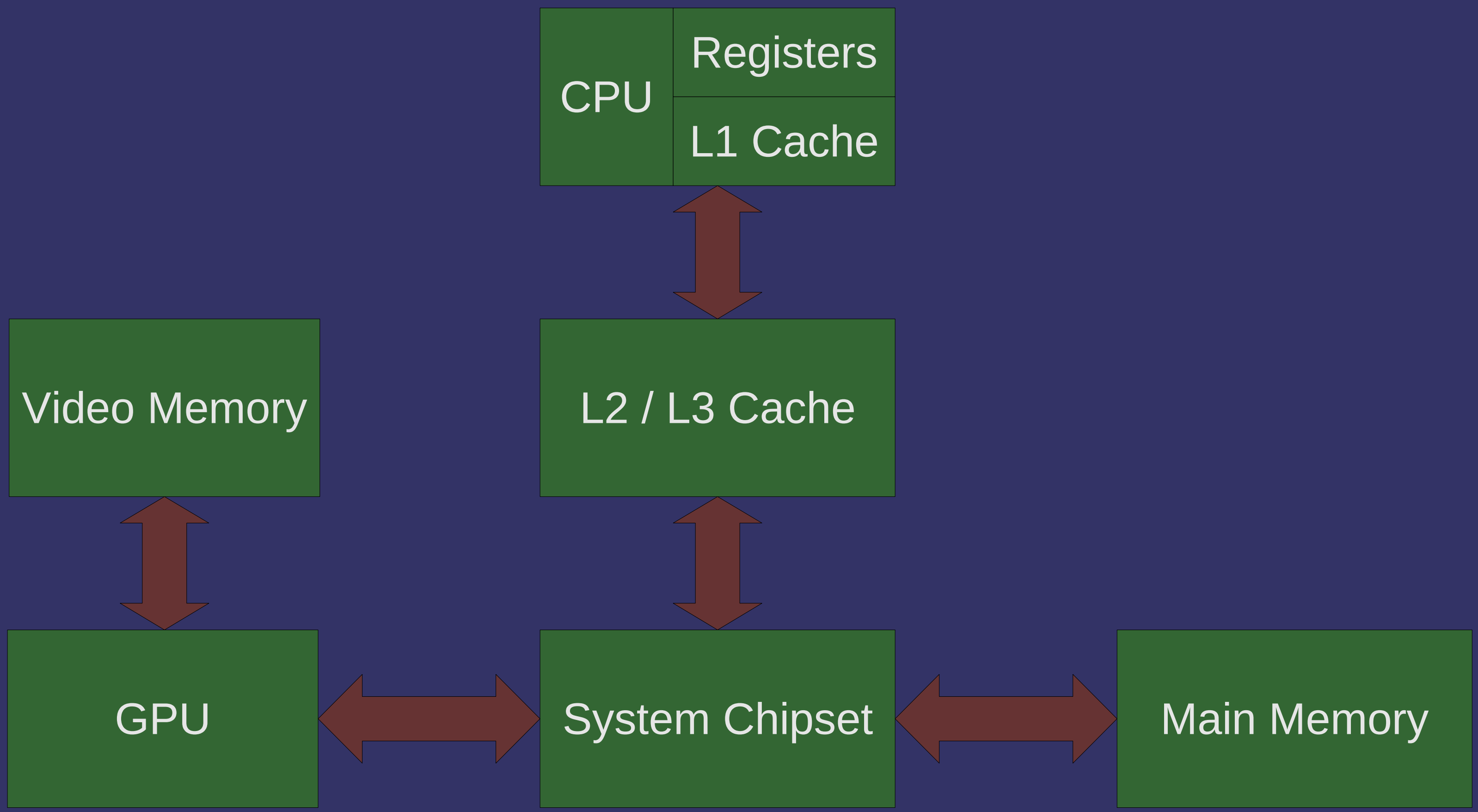
GPU accesses this data *directly*

Memory Architecture

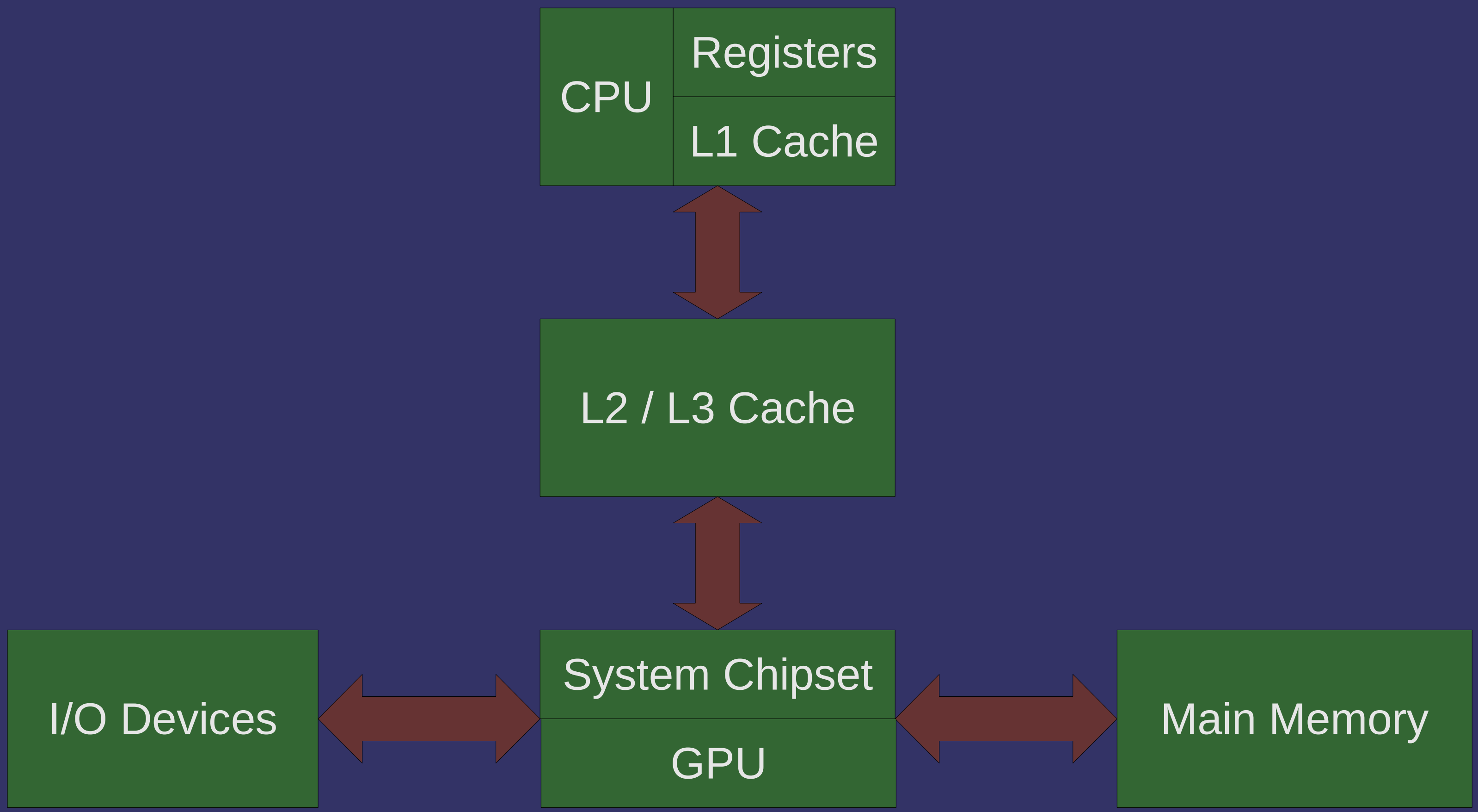


22-January-2009

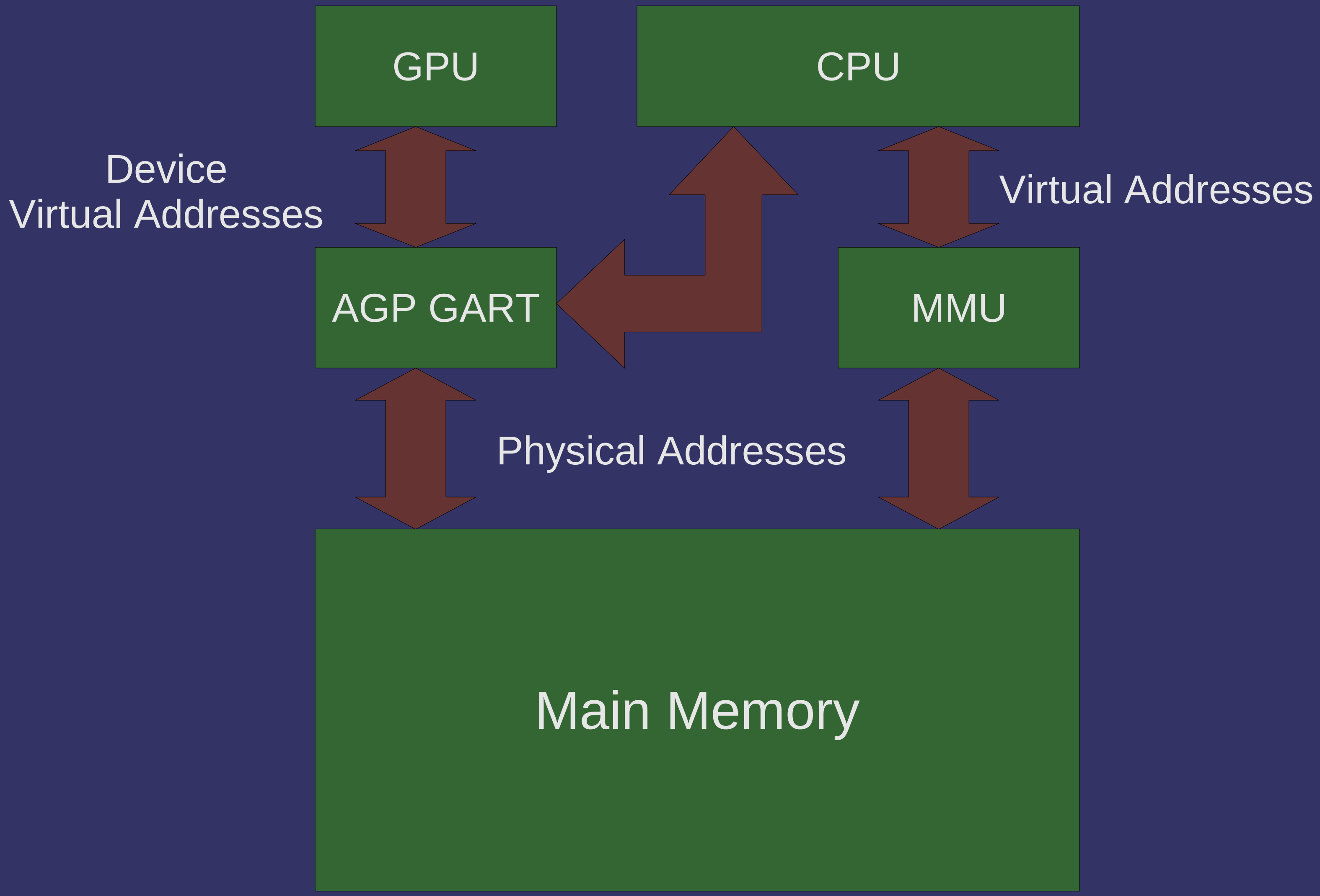
Memory Architecture



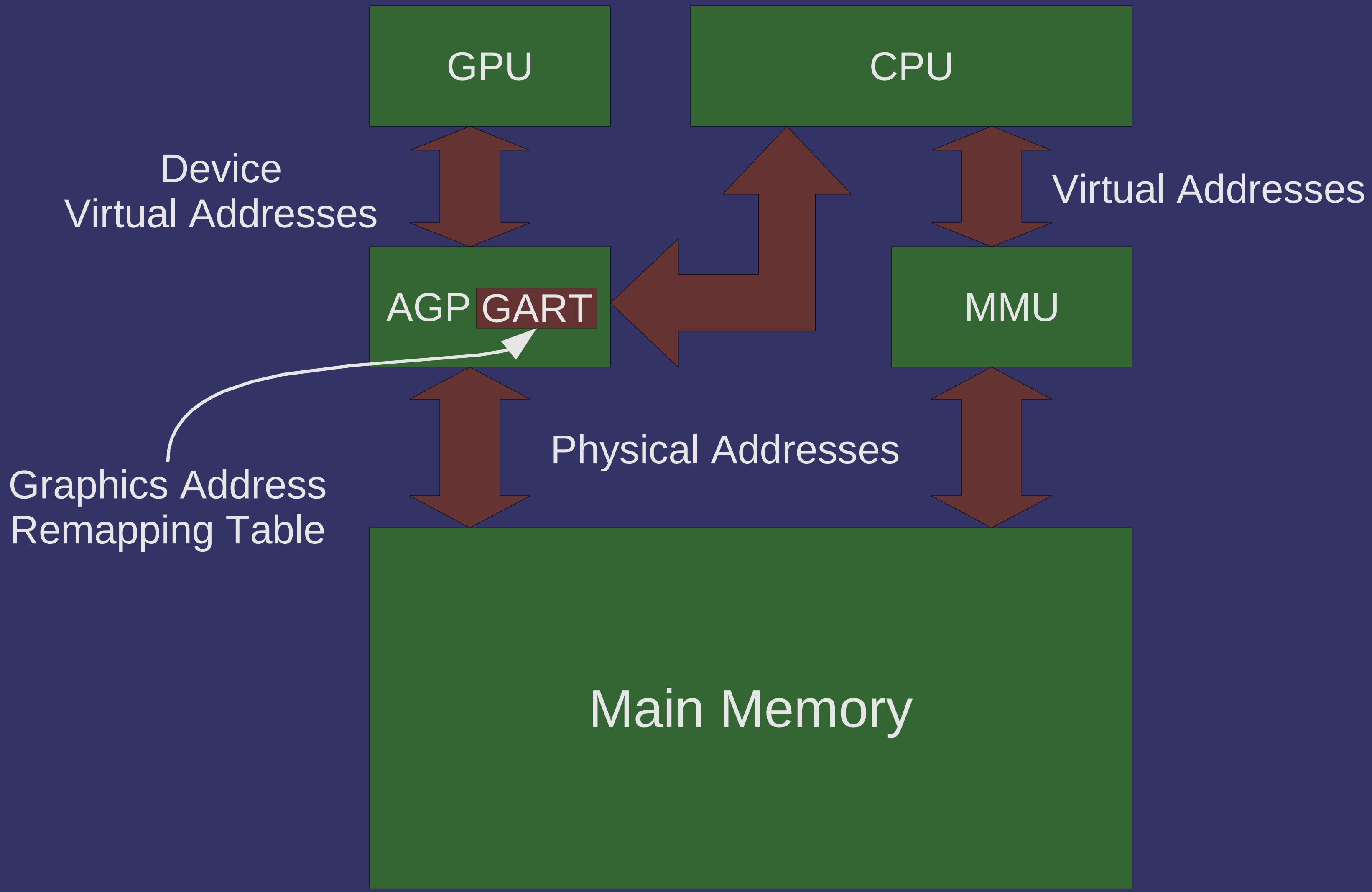
Unified Memory Architecture



Memory Map


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

Memory Map


GPU
CPU
Device
Virtual Addresses
Virtual Addresses
AGP GART
MMU
Physical Addresses
Graphics Address
Remapping Table
Main Memory

Vertex Memory

- Practically, the GPU can only access:
  - Memory physically on the graphics card
  - Memory mapped in the GART
- To get GART or card memory, we have to allocate it using the driver
  - Only the driver knows what *kind* of memory to use
  - ...but we have to give it some hints

Vertex Memory

- In OpenGL this memory is called *buffer object*
  - It is used somewhat like a file:
    - Bulk I/O via accessor routines
    - Direct mapping and access via a pointer

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 *later*...
    - There are other targets we'll cover later in the term
  - Binding creates the object, but it still has no storage

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 Objects

- Usage conveys information along two axes:
  - Data “frequency”:
    - Stream – data is specified once and used a few times
    - Static – data is specified ones 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 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 (GLint size)

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

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 *after* linking the program
  - See also `program::bind_attrib_location`

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

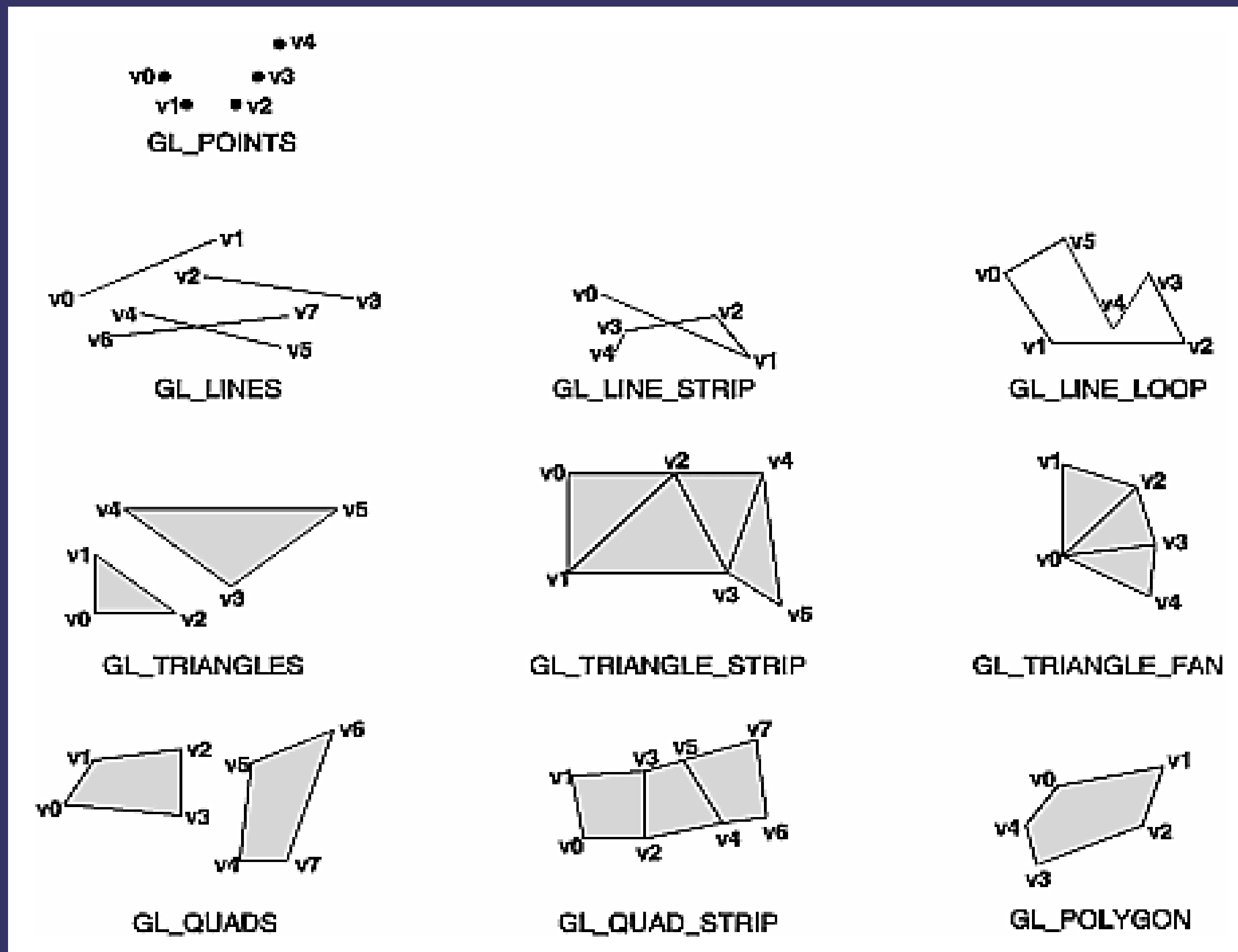


Image borrowed from "OpenGL Programming Guide".

Primitive Types

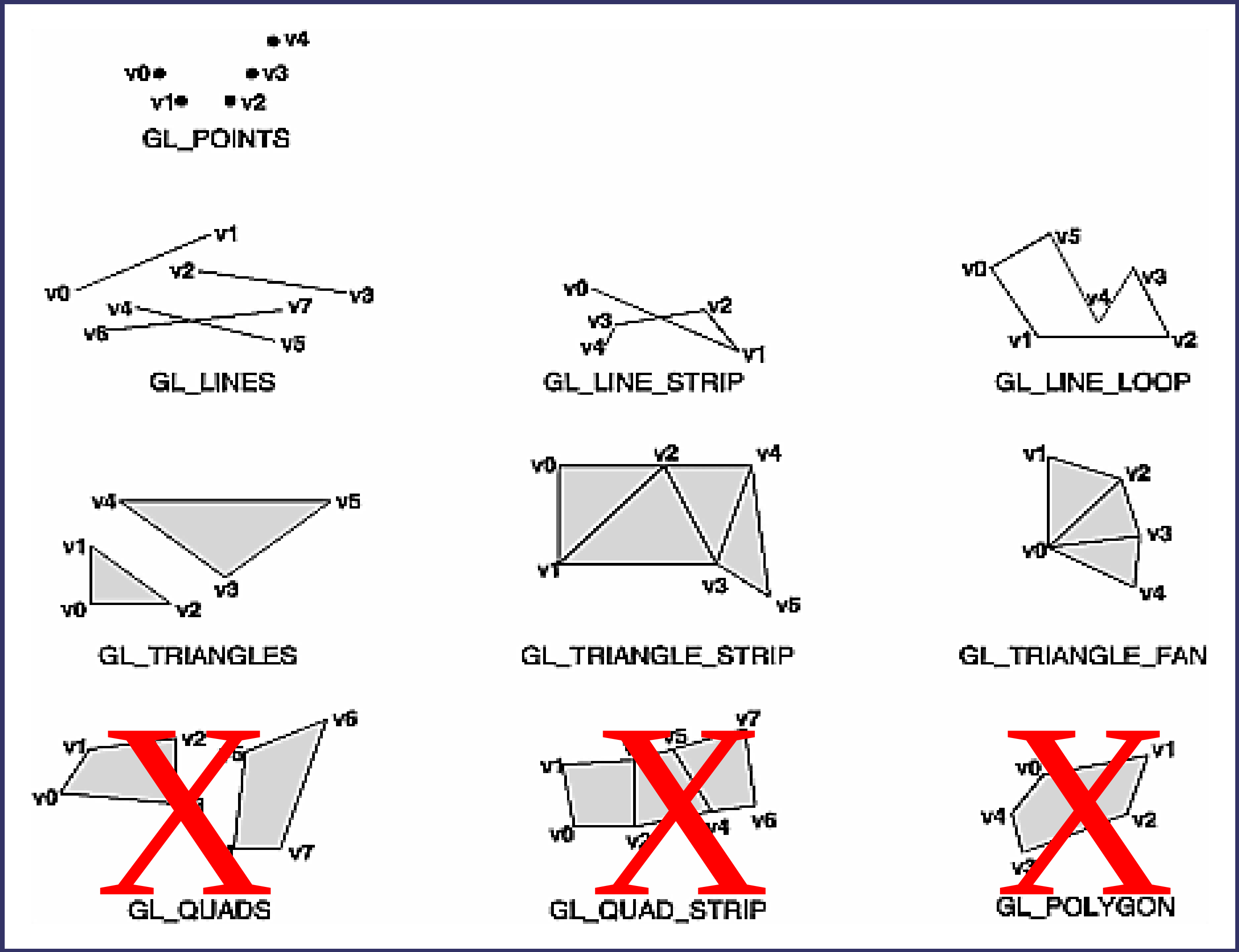


Image borrowed from “OpenGL Programming Guide”.

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

Break

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

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 T in the exponent
  - Example: V^{T}

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

- Noted as a “dot” between two vectors (e.g., $A \bullet B$)
  - Also known as the *inner product*
- Component-wise multiply, then sum components
  - Example:

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

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
    - $|A| = 1.0$
    - 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: $\frac{A}{|A|}$
  - 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: $A \bullet B = |A|\ |B| \cos \theta$
- We often want to know the angle between two vectors
  - This is the basis of all lighting calculations in 3D graphics!
  - $(A \bullet B) / (|A|\ |B|) = \cos \theta$

Cross Product

- From Wikipedia:

  In mathematics, the 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:

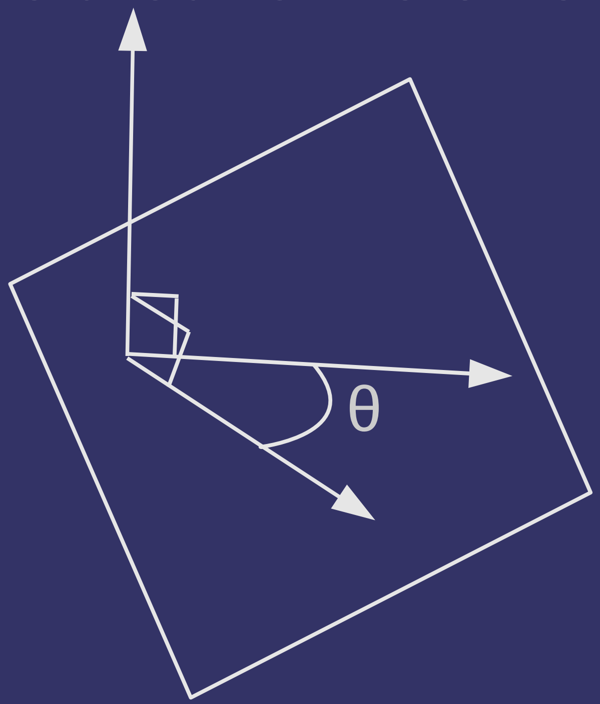
$$a \times b = \left[a_y b_z - a_z b_y \quad a_z b_x - a_x b_z \quad a_x b_y - a_y b_x \right]$$

  - Not associative
  - Anti-commutative: If $A \times B = C$, then $B \times A = -C$

[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: $A \times B = |A|\ |B| \sin\theta\ \mathrm{n}$



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., $M \times N \neq N \times M$
  - Associative e.g., $A(BC) = (AB)C$
  - Column count of first matrix must match row count of second matrix
    - If M is 4-by-3 matrix and N is a 3-by-1 matrix, we can do $M \times N$ but not $N \times M$
  - If the source matrices are n-by-m and m-by-p, the resulting matrix will be n-by-p

Matrix Multiplication

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

$$C_{ij} = \sum_{r=1}^{n} a_{ir} b_{rj}$$

- What does this look like?

Matrix Multiplication

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

$$C_{ij} = \Sigma_{r=1}^{n} a_{ir} b_{rj}$$

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

Multiplicative Identity

- There is a multiplicative identity for matrices

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

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

Transpose

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

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

References

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

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

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

Rotation

- Rotation around the Z-axis
  - If θ is 0, 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

- From the previous equations, we can rotate using 4 multiplies and 2 adds, but a matrix multiply requires 16 multiplies and 12 adds
  - $x' = x \cos \theta + y \sin \theta$
  - $x' = -x \sin \theta + y \cos \theta$
  - $z' = z$
- Why use the matrix method?

Rotation

- A series of rotations can be implemented as:

$$v' = M_1 v$$
$$v'' = M_2 v'$$
$$v''' = M_3 v''$$

- Which is the same as:

$$M_3(M_2(M_1 v))$$

- What can we do with this?

Rotation

- A series of rotations can be implemented as:

$$v' = M_1 v$$
$$v'' = M_2 v'$$
$$v''' = M_3 v''$$

- Which is the same as:

$$M_3(M_2(M_1 v))$$

- What can we do with this?

$$(M_3 M_2 M_1) v$$

  - Matrix multiplication is associative!

Arbitrary Rotation

- Given a vector, v, and an angle, θ, we can create an arbitrary rotation matrix:

$$\tilde{v} = \begin{bmatrix} 0 & -v_z & v_y & 0 \\ v_z & 0 & -v_x & 0 \\ -v_y & v_x & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R = \left(I * \cos\theta\right) - \left(\left(1 - \cos\theta\right) * \left(v * v^T\right)\right) + \left(\sin\theta * \tilde{v}\right)$$

Translation

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

$$p_x' = p_x M_{11} + p_y M_{12} + p_z M_{13} + p_w M_{14}$$
$$p_y' = p_x M_{21} + p_y M_{22} + p_z M_{23} + p_w M_{24}$$
$$p_z' = p_x M_{31} + p_y M_{32} + p_z M_{33} + p_w M_{34}$$
$$p_w' = p_x M_{41} + p_y M_{42} + p_z M_{43} + p_w M_{44}$$

- Since p_w is always 1, the 4^{th} column of the matrix acts as a translation

Scaling

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

$$M = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Coordinate Spaces

- Coordinates are always relative to some “space”
  - Object space: Local coordinate system of the object
  - World space: Global coordinate system relative to the 3D “world”
  - Eye / camera space: Coordinate system relative to the viewer
- When we translate objects relative to other objects, we may talk about other spaces
  - If the hand of a 3D model is rotated relative to the arm of the model, we may talk about “hand-space” or “arm-space”

Orthonormal Basis

- It's a mouthful...what does it mean?
- A vector space where all of the components are *orthogonal* to each other, and each is *normal*
  - Normal meaning unit length
  - Orthogonal meaning at right angles
    - The *other* meaning of normal
- Every pure rotation matrix (i.e., no scaling) is orthonormal basis
  - As is the identity matrix

Viewing

- Q: Given a world position for a camera, a world position to point the camera at, and an “up” direction, how can we construct a transformation using just rotations and translations?

Viewing

- Q: Given a world position for a camera, a world position to point the camera at, and an “up” direction, how can we construct a transformation using just rotations and translations?
- A: We can't. We can construct an orthonormal basis from those 3 vectors

Viewing

- Given:
  - E: Position of the eye (or camera) in world-space
  - V: The point being viewed
  - U: the “up” direction
- Calculate the unit vector from the viewpoint to the eye:

$$F = E - V$$

$$f = \frac{F}{|F|}$$

  - This is the Z axis

Viewing

- Calculate a vector orthogonal to the Z-axis and the up vector:

$$s = f \times u$$

  - This is the X-axis

Viewing

- Calculate a vector orthogonal to the Z-axis and the up vector:

$$s = f \times u$$

  - This is the X-axis
- Calculate a vector orthogonal to the X-axis and the Z-axis:

$$t = f \times s$$

  - This is the Y-axis
  - Why can't we just use U?

Viewing

- Drop these vectors into a matrix:

$$M_v = \begin{bmatrix} s_0 & s_1 & s_2 & -E_0 \\ t_0 & t_1 & t_2 & -E_1 \\ f_0 & f_1 & f_2 & -E_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

  - The translation moves the eye to the origin

References

General information about rotation matrices and orthonormal bases:

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

http://www.wikipedia.org/Orthonormal_basis

Really good explanation of arbitrary rotation matrices:

http://www.euclideanspace.com/maths/geometry/rotations/conversions/angleToMatrix/index.htm

Projection

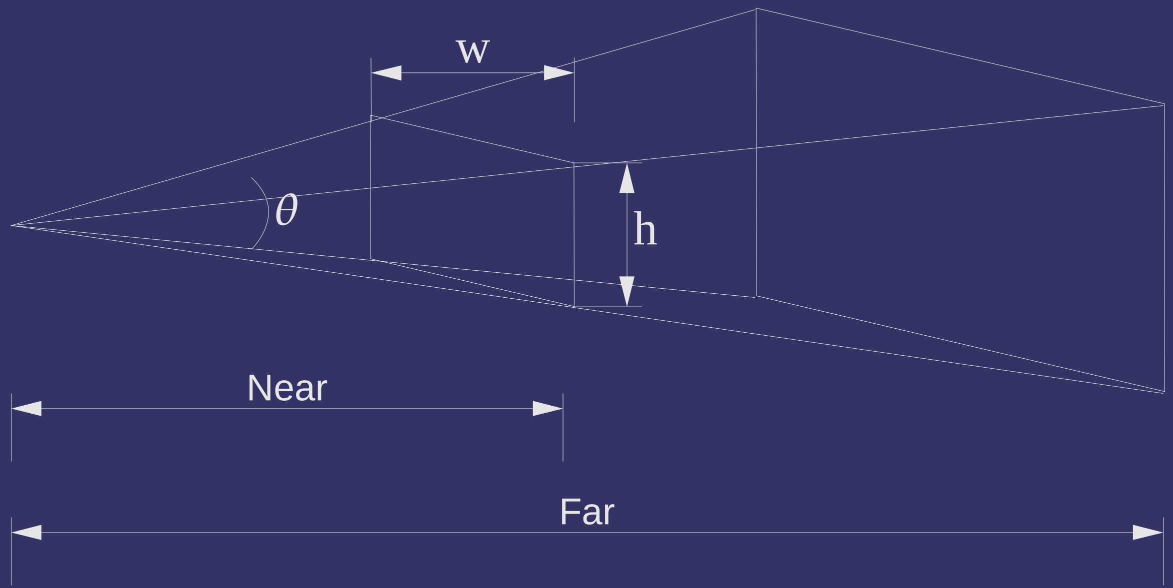
- Once objects are transformed to camera-space, they're still 3D
  - The screen is still 2D
  - Camera parameters (e.g., field of view) need to be applied
- Two steps remain:
  - Projection from camera space to screen space
  - Perspective divide

Projection

- Perspective:
  - Simulates visual foreshortening caused by the way light projects onto the back of the eye
  - Represents the view volume with a frustum (a pyramid with the top cut off)
  - The real work is done by dividing X and Y by Z
- Orthographic:
  - Represents the view volume with a cube
  - Also called *parallel projection* because lines that are parallel before the projection remain parallel after

Perspective Projection

- A few parameters control the view volume:
  - Near: Distance from the camera to the near viewing plane. Objects in front of this plane will be clipped
  - Far: Distance from the camera to the far viewing plane. Objects behind this plane will be clipped
  - θ: Field-of-view in the Y direction
  - Aspect ratio: Ratio of the width of the view to the height of the view



Perspective Projection

$$f = \cot\left(\frac{\theta}{2}\right)$$

$$M_p = \begin{bmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{far + near}{near - far} & \frac{2 \times far \times near}{near - far} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Putting it all together

- Typically have a *modeling* transform, a *viewing* transform, and a *projection*
  - Combine these into a single “modelviewprojection” matrix: $M_{\mathrm{mvp}} = M_{\mathrm{p}} \times M_{\mathrm{v}} \times M_{\mathrm{m}}$
  - Transform a vertex by this single matrix:

```
uniform mat4 mvp;
void main(void)
{
        gl_Position = mvp * gl_Vertex;
}
```

References

http://en.wikipedia.org/wiki/3D_projection (esp. Third step: perspective transform).

http://en.wikipedia.org/wiki/Orthographic_projection_%28geometry%2

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

Next week...

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

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.