Basics for Enhanced Visualization: 3D/Data

3D Rendering

Rodrigo Cabral

Polytech Nice - Data Science

cabral@unice.fr

Source: by Gilles Tran - http://www.oyonale.com/modeles.php?lang=en&page=40, Public Domain, https://commons.wikimedia.org/w/index.php?curid=745313.

Outline

1. Introduction — Class 1
2. OpenGL rendering pipeline
3. Geometric primitives
4. Tessellation
5. Simple drawing in OpenGL
6. Transformations

---

7. Rasterization — Class 2
8. Viewport
9. Image formation in OpenGL
10. Occluded objects and the Z-buffer
11. Conclusions

These slides are partly inspired from Lionel Fillatre's slides.

Introduction

Augmented reality main steps:

1. Detect a triggering pattern for augmentation: **image processing and machine learning**.

Augmented reality main steps:

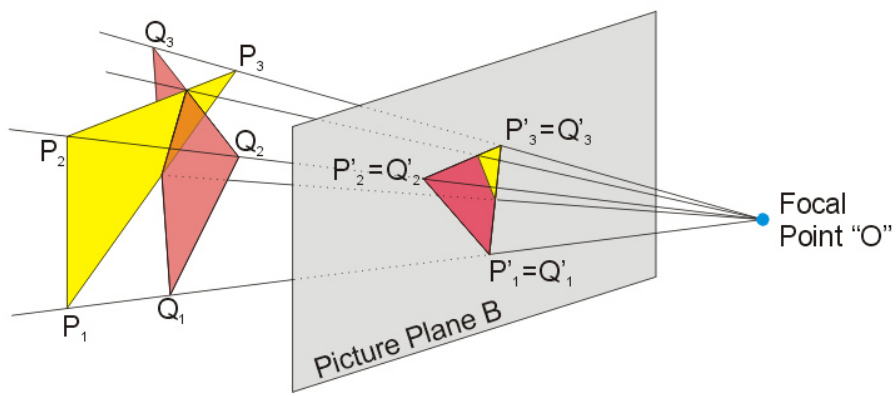
1. Detect a triggering pattern for augmentation: **image processing and machine learning**.
2. Get 3D scene structure: **3D computer vision**.
   Basics on the last classes.
3. Real-time Insertion of a realistic projection of a 3D object on the image buffer: **real-time 3D computer graphics**.
   Basics on this and next classes.

Augmented reality main steps:

1. Detect a triggering pattern for augmentation: **image processing and machine learning**.
2. Get 3D scene structure: **3D computer vision**.
   Basics on the last classes.
3. Real-time Insertion of a realistic projection of a 3D object on the image buffer: **real-time 3D computer graphics**.
   Basics on this and next classes.
4. Track triggering pattern and scene structure: **signal processing**.
5. Update projection of the virtual object.

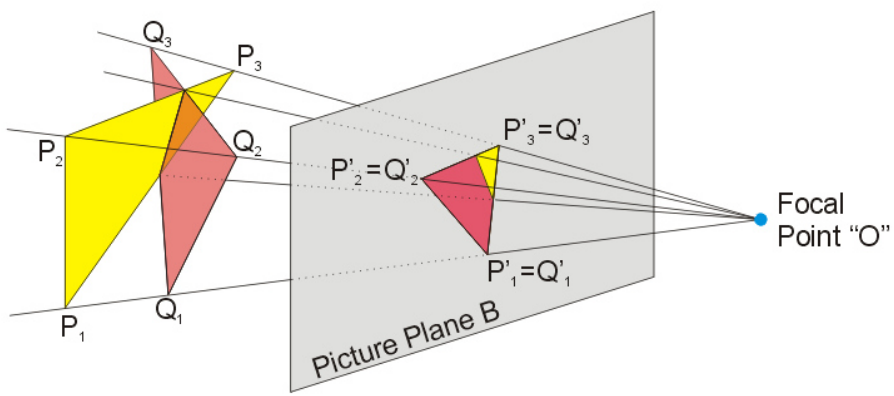
Requirements for 3D real-time rendering:

- Handle the projection of hundreds, thousands or millions of points on the image buffer.



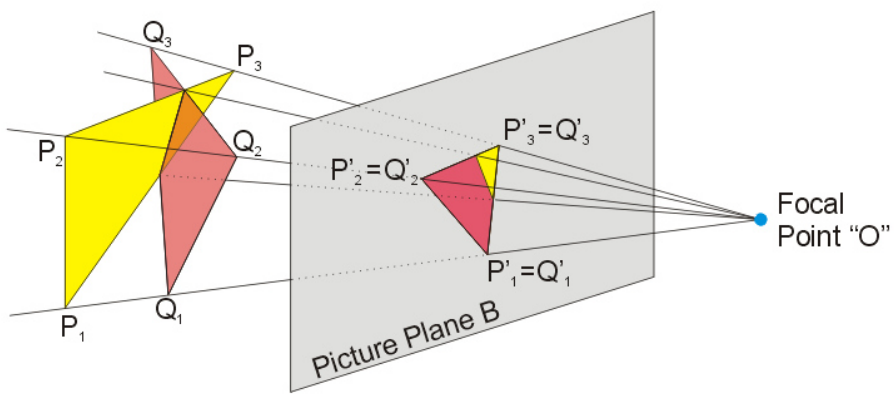
Requirements for 3D real-time rendering:

- Handle the projection of hundreds, thousands or millions of points on the image buffer.
- Evaluate how to fill the RGB image buffers between these points: color, lighting, texture, material...



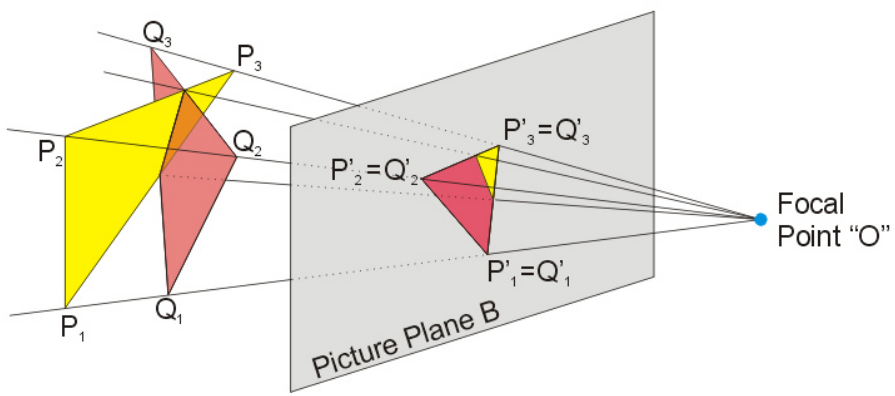
Requirements for 3D real-time rendering:

- Handle the projection of hundreds, thousands or millions of points on the image buffer.
- Evaluate how to fill the RGB image buffers between these points: color, lighting, texture, material...
- Check which parts of the projection should appear in the image: occlusions!



Requirements for 3D real-time rendering:

- Handle the projection of hundreds, thousands or millions of points on the image buffer.
- Evaluate how to fill the RGB image buffers between these points: color, lighting, texture, material...
- Check which parts of the projection should appear in the image: occlusions!
- Do all this at a rate greater than 12 images per second: requirement for tricking the human eye.



3D real-time rendering:

- This was a **Big Data issue**: the processors were not capable of dealing with all these data in real-time.

Source: Swaaye at the English language Wikipedia, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=6037820

Introduction

3D real-time rendering:

- This was a **Big Data issue**: the processors were not capable of dealing with all these data in real-time.
- The gaming industry soon noted this issue: growth in polygons per/second were bounded by the processor capability.

$$\text{PS} : 9 \times 10^4 \text{ poly./sec.} \qquad \text{N64} : 16 \times 10^4 \text{ poly./sec.}$$

$$\text{PS2} : 15 \times 10^6 \text{ poly./sec.}$$

Source: Swaaye at the English language Wikipedia, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=6037820

Introduction

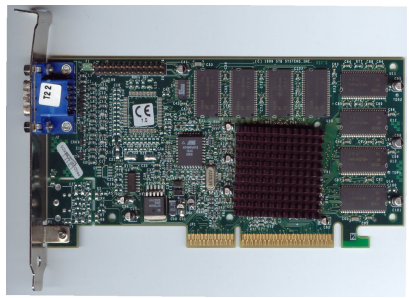
3D real-time rendering:

- This was a **Big Data issue**: the processors were not capable of dealing with all these data in real-time.
- The gaming industry soon noted this issue: growth in polygons per/second were bounded by the processor capability.

$$\text{PS} : 9 \times 10^4 \text{ poly./sec.} \qquad \text{N64} : 16 \times 10^4 \text{ poly./sec.}$$

$$\text{PS2} : 15 \times 10^6 \text{ poly./sec.}$$

A solution appeared: create a specialized powerful hardware for graphics processing$\Longrightarrow$ **graphics processing unit (GPU)**.



Source: Swaaye at the English language Wikipedia, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=6037820

Introduction

3D real-time rendering:

- How to give instructions to the graphics card?
- **Open Graphics Library (OpenGL)**

Source: Crysis, Crytek/EA, 2007.

OpenGL rendering pipeline

OpenGL

- OpenGL primary function: rendering
- What is rendering? Converting geometrical/mathematical object descriptions into frame buffer values.
- In computer graphics the **image buffer is called frame buffer**.

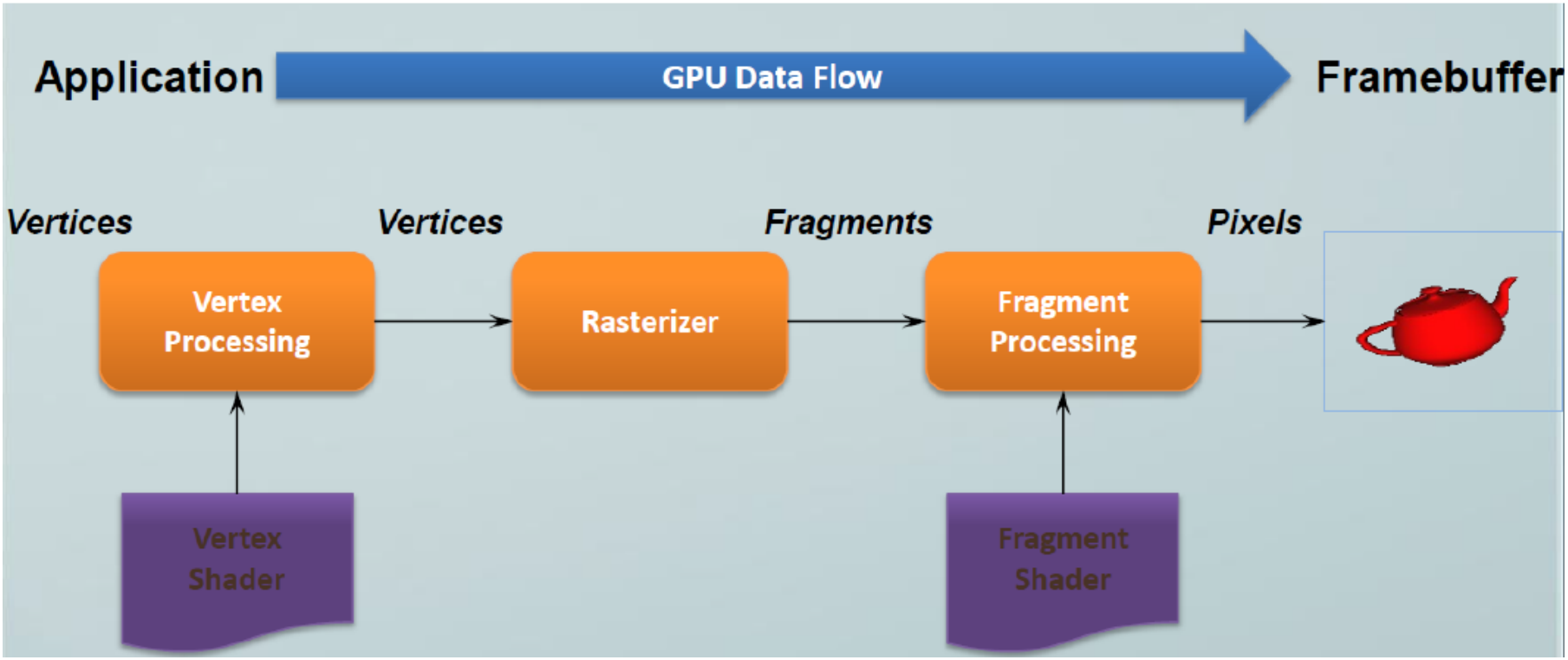
OpenGL rendering pipeline

OpenGL

- OpenGL primary function: rendering
- What is rendering? Converting geometrical/mathematical object descriptions into frame buffer values.
- In computer graphics the **image buffer is called frame buffer**.
- OpenGL can render
  - geometric primitives (we will see then later),
  - raster primitives: bitmaps and images.

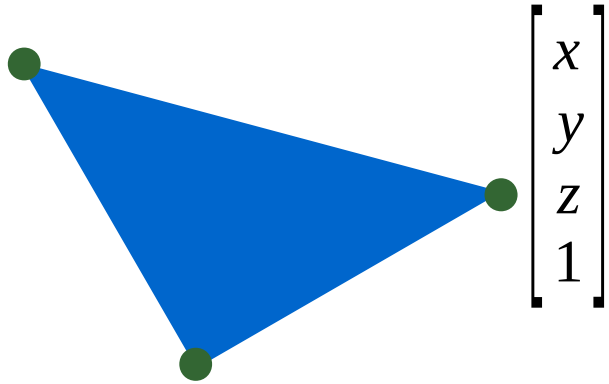
OpenGL rendering pipeline

Rendering pipeline



Geometric objects

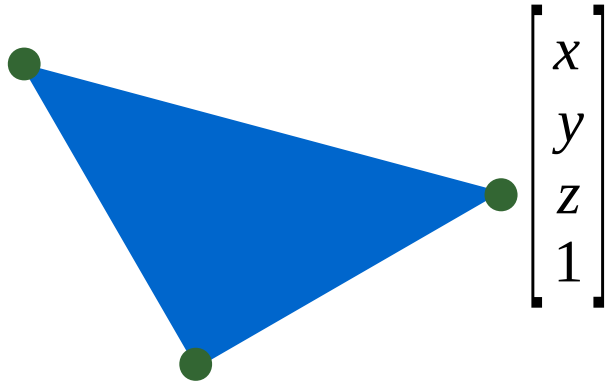
- Geometric objects are represented using vertices.



- A vertex is a collection of generic attributes:
  - positional coordinates,
  - colors,
  - texture coordinates,
  - any other data associated with points in space

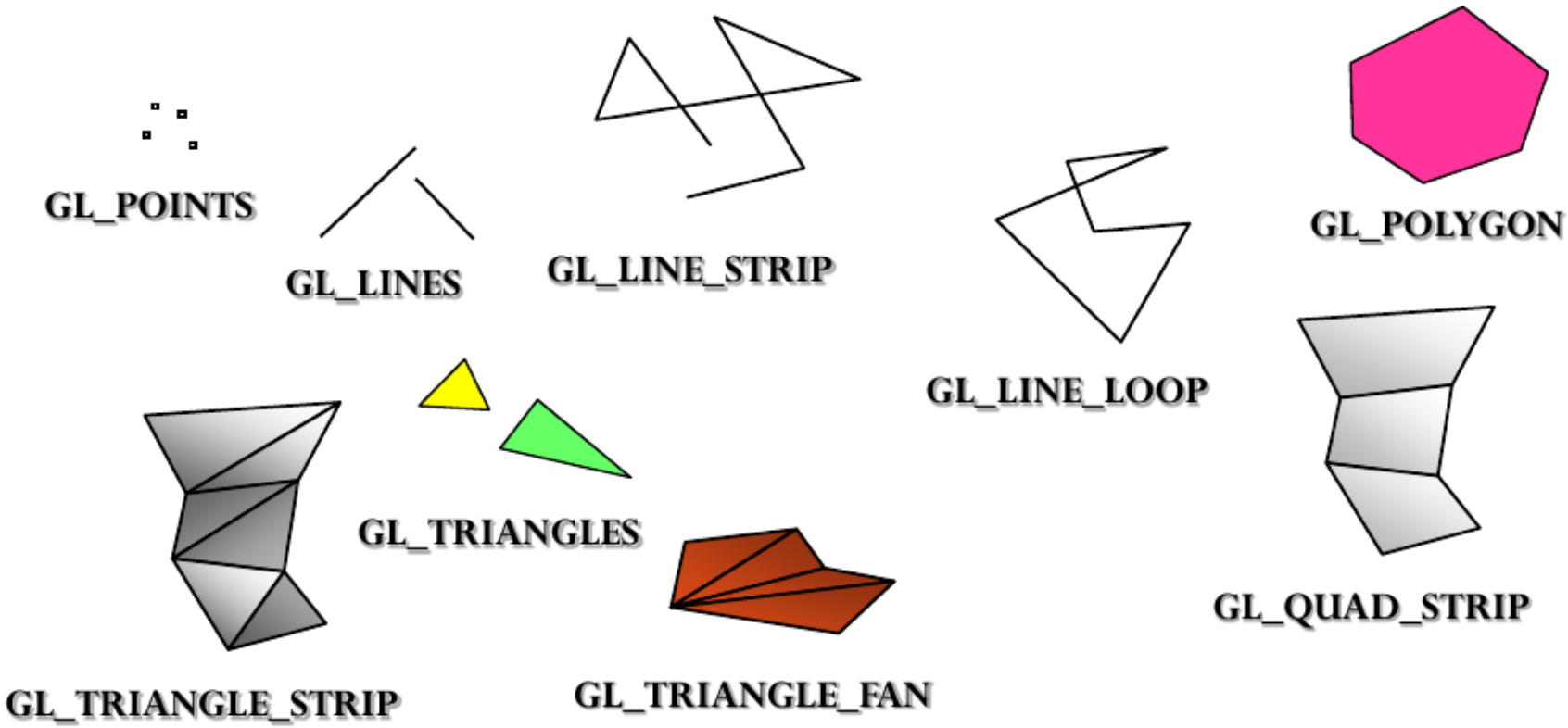
Geometric objects

- Geometric objects are represented using vertices.



- A vertex is a collection of generic attributes:
  - positional coordinates,
  - colors,
  - texture coordinates,
  - any other data associated with points in space
- Position is stored in homogeneous coordinates.
- Vertex of an object are stored in vertex buffer objects (VBO).
- VBO are stored in vertex array objects (VAO).

Geometric primitives



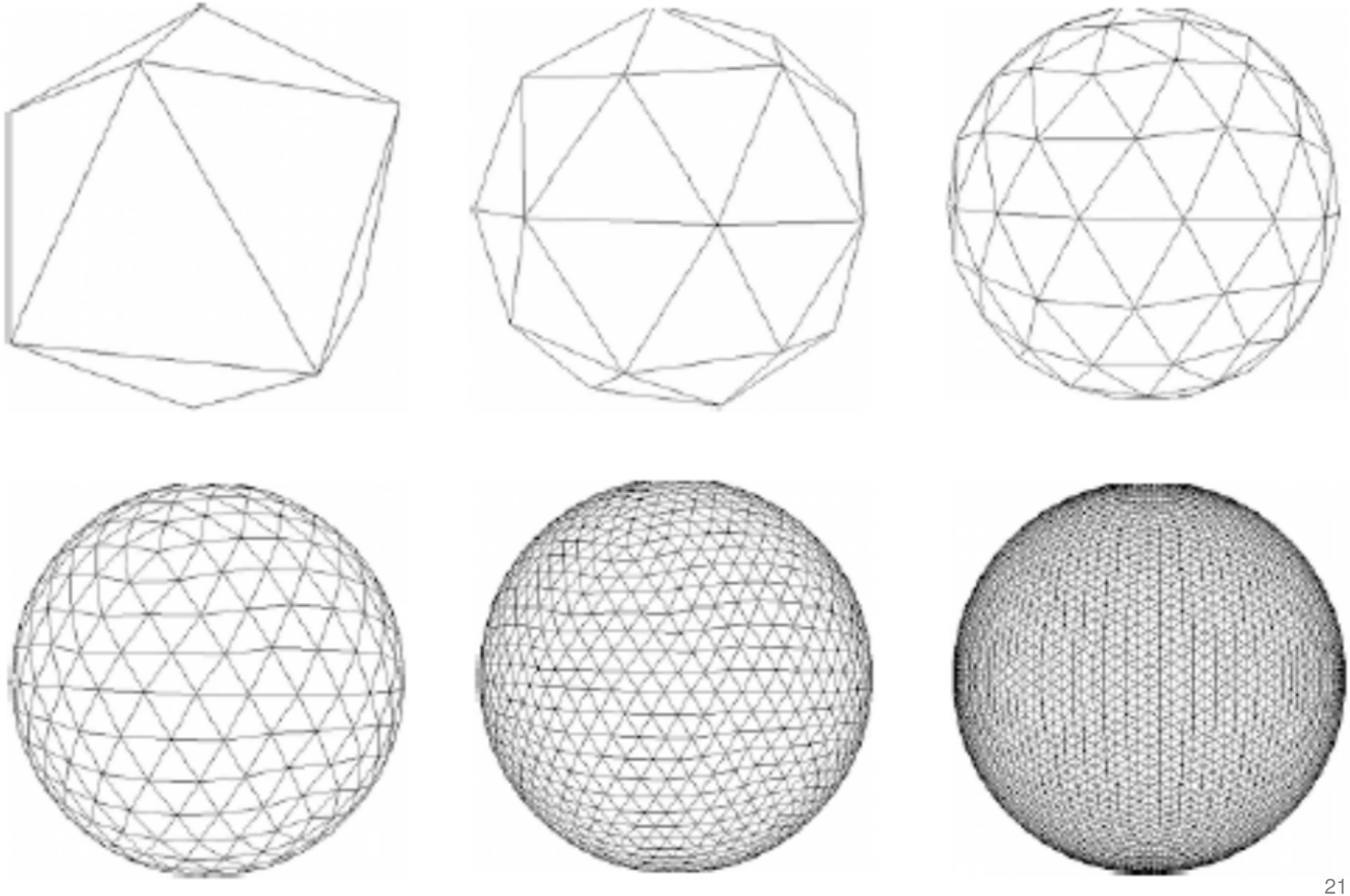
Geometric primitives

Basic OpenGL

- Represent object surface as a set of primitive shapes
  - points,
  - lines,
  - triangles,
  - quad(rilateral)s
- This process is called **tessellation**.
- Draw primitive one by one.
  - Batched and parallelized in hardware.

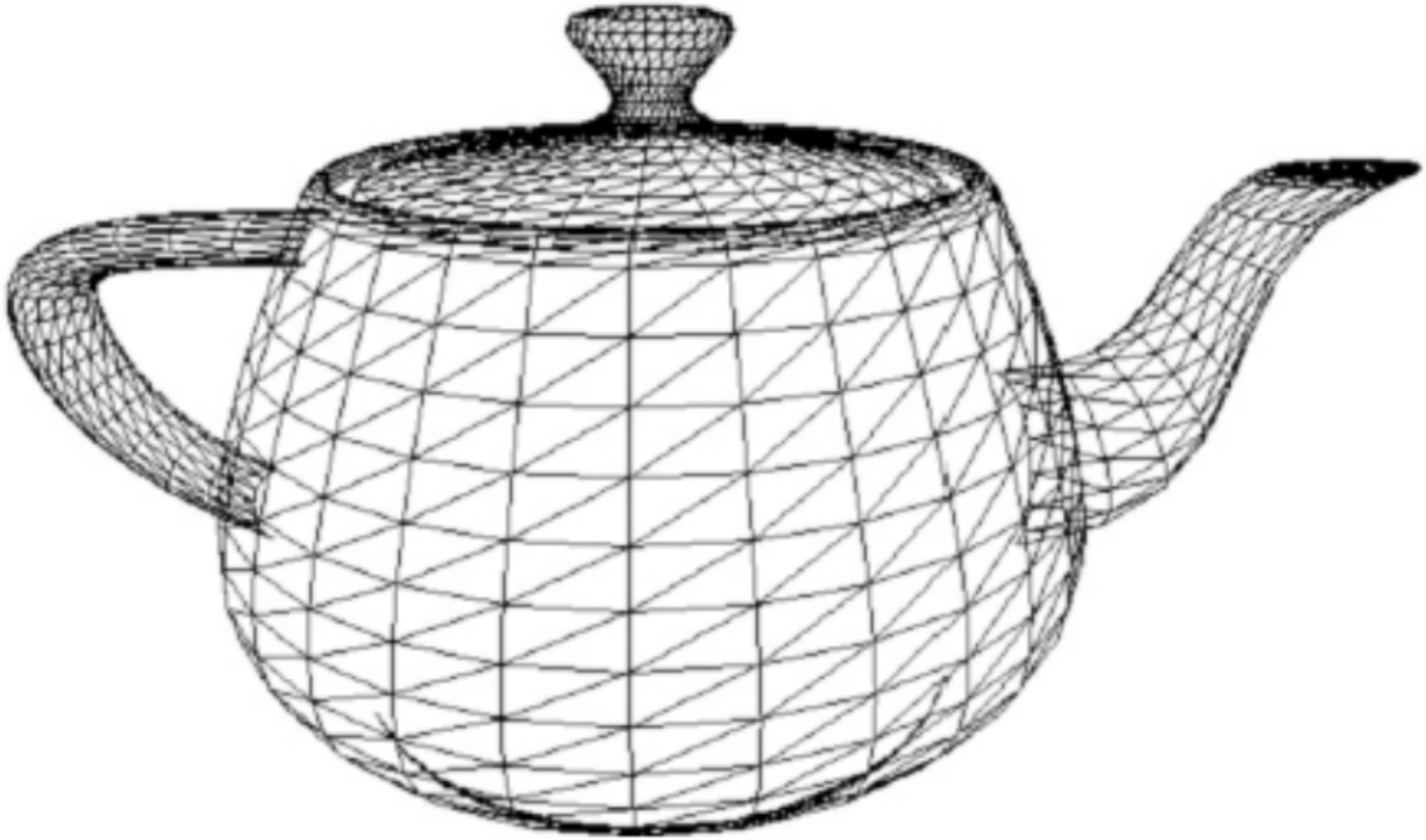
Tessellation

Tessellating a sphere with triangles



Tessellation

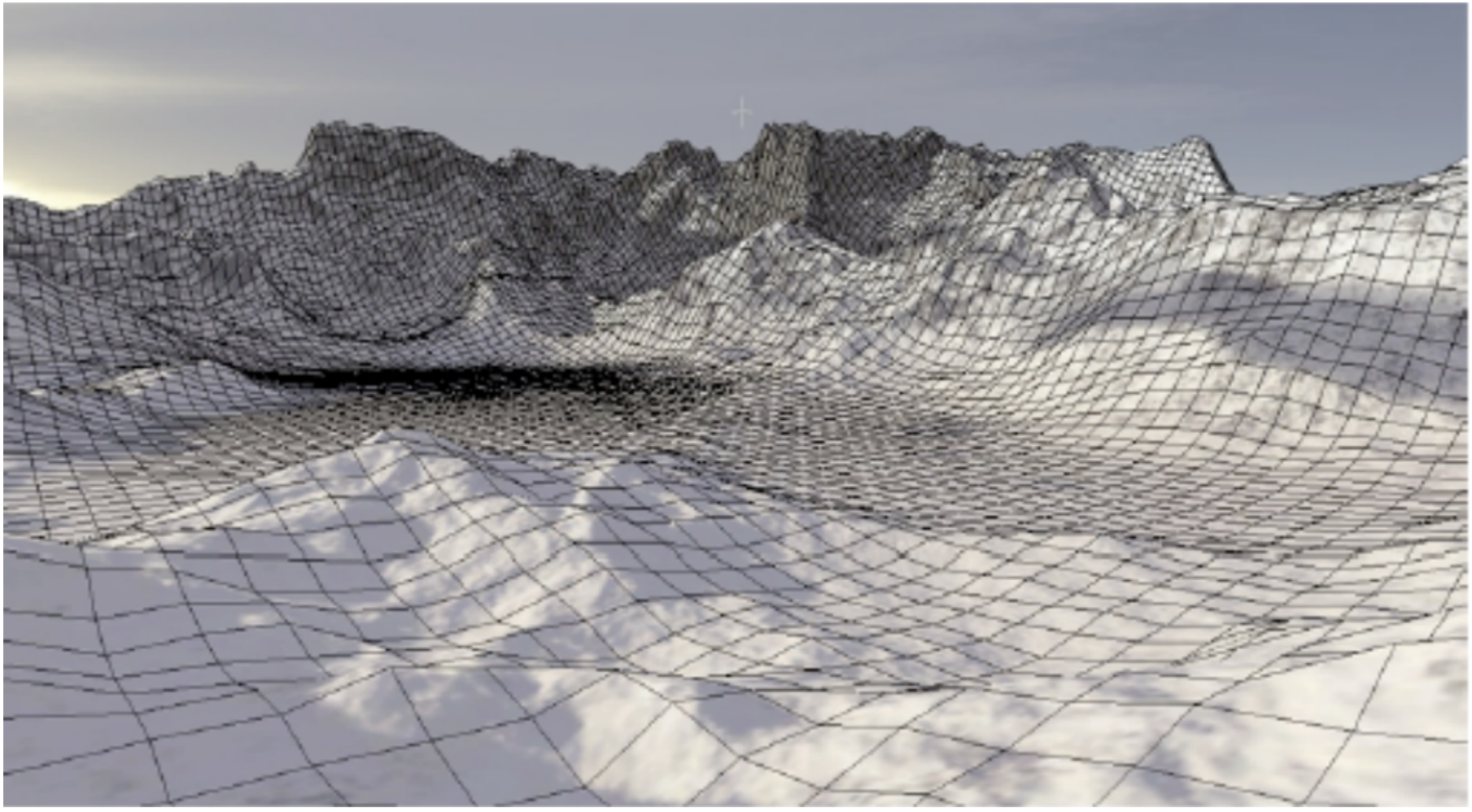
Tessellating a teapot with triangles



Tessellating animals with triangles

Tessellation

Tessellating terrain with triangles



Tessellation

- Difficult to get right:
  - Primitives must be evenly distributed.
  - Primitives must not have awkward shapes (e.g. very skinny triangles).
  - This is important not only for display, but even more for physics simulation methods (e.g. finite elements).
- Many sophisticated algorithms exist:
  - Often take equations of curved patches as input.
  - These algorithms will not be presented on this course.

Drawing triangles in OpenGL

```
# Draw a triangle
glBegin(GL_TRIANGLES)                    # Start drawing a triangle
glVertex3f(0.0, 1.0, 0.0)                # Top
glVertex3f(1.0, -1.0, 0.0)               # Bottom Right
glVertex3f(-1.0, -1.0, 0.0)              # Bottom Left
glEnd()
```

- Every collection of primitives must be placed between **glBegin(primitive)** and **glEnd()**.
- Every three successive vertices in this example will define a triangle.
- Instead of the primitive `GL_TRIANGLES` we could use `GL_POINTS` (vertices are points) or `GL_LINES` (every 2 vertices define a line) or `GL_QUADS` (every 4 vertices define a quadrilateral) *etc*.

Drawing triangles in OpenGL

```
# Draw a triangle
glBegin(GL_TRIANGLES)          # Start drawing a polygon

glNormal3f(0.58,0.58,0.58)     # Normal vector to the triangle
glColor3f(1.0,0.0,0.0)         # Color (red)

glVertex3f(1.0, 0.0, 0.0)      # Top
glVertex3f(0.0, 1.0, 0.0)      # Bottom Right
glVertex3f(0.0, 0.0, -1.0)     # Bottom Left
glEnd()                        # We are done with the polygon
```

- We set the normal and color per triangle. They can actually be set anywhere, anytime and be applied to all subsequent vertices.

Simple drawing in OpenGL

What is ...3f?

- **glVertex** has variants glVertex3f and glVertex3d.
  - The first takes 3 float arguments (x, y, z).
  - The second takes 3 doubles.
  - OpenGL has also ...3i functions which takes integers.
- There is also glVertex2f, where z is assumed to be zero.
- And glVertex4f, where last argument is homogeneous coordinate w, otherwise assumed to be 1.
- Similarly glColor4f is used to specify (R, G, B, α).

Transforming objects: a simple example

```
M=np.array([[m_11,m_12,m_13,m_14],
            [m_21,m_22,m_23,m_24], # Matrix of transformation
            [m_31,m_32,m_33,m_34], # in homogeneous coordinates
            [m_41,m_42,m_43,m_44]])
M=M.T                              # These two lines transform
M=M.flatten()                      # transform matrix in vector

# Tranformation to be applied to the object
glLoadMatrixf(M)

# Draw objects
glBegin(Primitive)  # Draw primitive and then
.                   # apply transformation M
.
.
glEnd()
```

- The object is transformed by **M** before its drawn: each vertex **v** is transformed with **Mv**

Transforming objects: a simple example

```
M=np.array([[m_11,m_12,m_13,m_14],
            [m_21,m_22,m_23,m_24], # Matrix of transformation
            [m_31,m_32,m_33,m_34], # in homogeneous coordinates
            [m_41,m_42,m_43,m_44]])
M=M.T                              # These two lines transform
M=M.flatten()                      # transform matrix in vector

# Tranformation to be applied to the object
glLoadMatrixf(M)

# Draw objects
glBegin(Primitive)   # Draw primitive and then
.                    # apply transformation M
.
.
glEnd()
```

- The matrix **M** is given in vectorized version, with vectorization by columns.

Transforming objects: compositions

```
glLoadMatrixf(M_1)
glLoadMatrixf(M_2)
glLoadMatrixf(M_3)
```

- Subsequent object vertices will be transformed by matrix $\mathbf{M} = \mathbf{M}_1 \mathbf{M}_2 \mathbf{M}_3$.
- **Transformations are applied last-to-first.**

Transforming objects: OpenGL functions

- glLoadIdentity() ≡ glLoadMatrix(**I**).
- glTranslatef(t_x,t_y,t_z) ≡ glMultMatrix(**T**).
  - **T** is a transformation matrix that transforms by $[t_x\ t_y\ t_z]^T$
- glRotatef(θ,u_x,u_y,u_z) ≡ glMultMatrix(**R**).
  - **R** is a transformation matrix that rotates vectors by an angle θ around vector $[u_x\ u_y\ u_z]^T$.
- glScalef(s_x,s_y,s_z) ≡ glMultMatrix(**S**).
  - **S** is a transformation matrix that scales vectors by s_x on the x axis, s_y on the y axis and s_z on the z axis.
- All this functions have ...d versions.

Transforming objects: a more complicated example

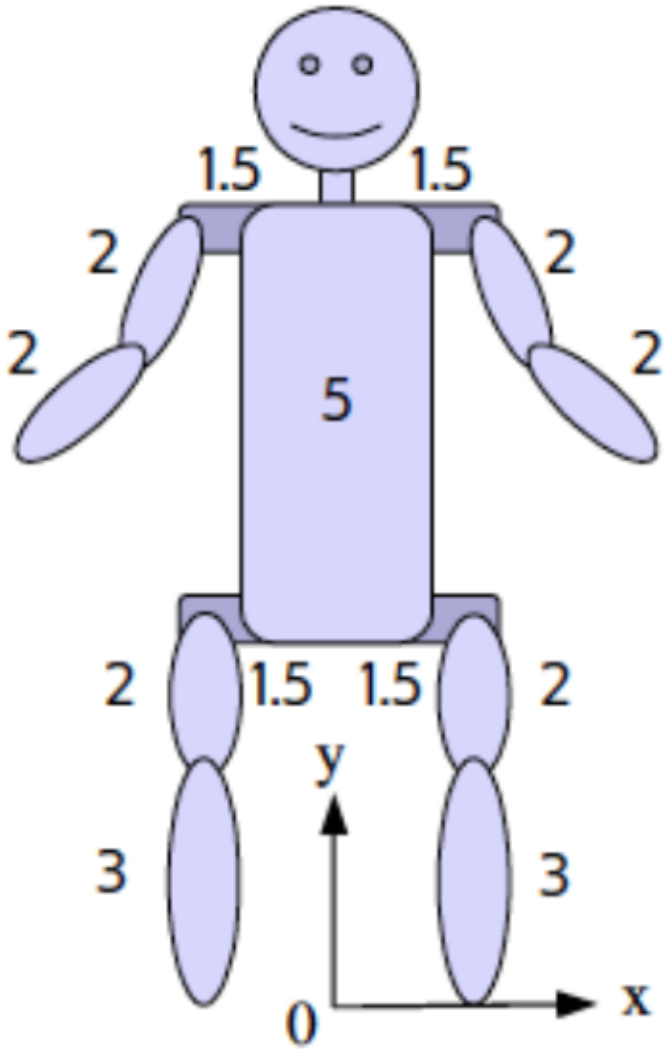
```
glMatrixMode(GL_MODELVIEW)

glPushMatrix()
glMultMatrix(M)

# Draw objects
glBegin(Primitive)   # Draw primitive and then
.                    # apply transformation M
.
.
glEnd()
glPopMatrix()
```

- Push/Pop?
- What is MatrixMode?

Hierarchical modeling

- Computer graphics systems maintain a current transformation matrix (CTM).
- All geometry is transformed by CTM.
- CTM defined object space where all geometry is specified.
- Transformation commands are concatenated in the CTM. The last one added is applied first:
  - CTM := CTM **T**
- CTM can be reset with **glLoadMatrix()**.
- Computer graphics systems also maintain a transformation stack:
  - CTM can be pushed (saved) onto the stack with glPushMatrix().
  - CTM can be popped (loaded) from the stack with glPopMatrix().

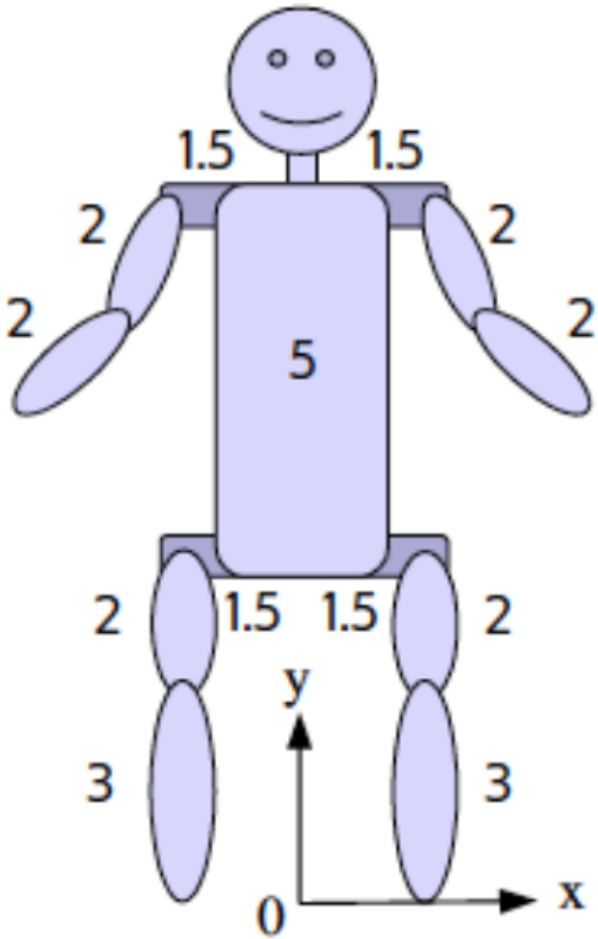
Example: articulated robot



- body
  - torso
    - head
    - shoulder
      - leftArm
        - upperArm
        - lowerArm
        - hand
      - rightArm
        - upperArm
        - lowerArm
        - hand
    - hips
      - leftLeg
        - upperLeg
        - lowerLeg
        - foot
      - rightLeg
        - upperLeg
        - lowerLeg
        - foot

Transformations

Example: articulated robot




```
glTranslatef(0, 1.5, 0)
drawTorso()
glPushMatrix()
glTranslatef(0, 5, 0)
        drawShoulder()
        glPushMatrix()
                glRotatef(neck_y, 0, 1, 0)
                glRotatef(neck_x, 1, 0, 0)
                drawHead()
        glPopMatrix()
        glPushMatrix()
                glTranslatef(1.5, 0, 0)
                glRotatef(l_shoulder_x)
                drawUpperArm()
                glPushMatrix()
                        glTranslatef(0,-2,0)
                        glRotatef(l_elbow_x, 1, 0, 0)
                        drawLowerArm()
                        ...
                glPopMatrix()
        glPopMatrix()
        ...
```

Summary

- Object surfaces are tessellated into simple primitives.
- Draw primitives with glBegin()/glEnd() blocks.
  - glVertexf() , glColorf() , glNormalf().
- Nested transformation blocks:
  - glPushMatrix() , glPopMatrix() , glLoadMatrix() , glMultMatrix().
- **Next class:** rasterization, view port, image formation in OpenGL(Model View mode) and the Z-buffer.