

# Week 10

Texture Mapping 1	▼
Texture Mapping 2	▼
Texture Mapping 3	▼
Hierarchical Modelling	▼
Hierarchical Modelling with Trees	▼
Generalizing Hierarchical Modelling	▼

# Texture Mapping 1

Wednesday, November 20, 2019 3:19 PM

There are **3 steps to applying a texture**

1. Specify the texture

- Read or generate image
- Assign image to texture
- Enable texture mapping in OpenGL

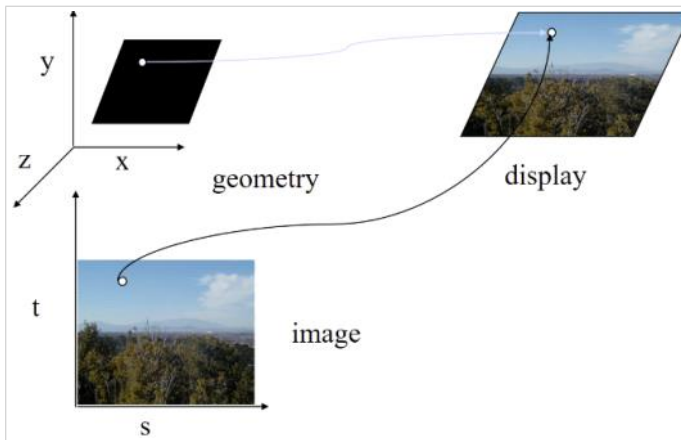
2. Assign texture coordinates to vertices (mapping)

- Proper mapping function is left to application
- We have to write code to assign texture coordinates to vertex coordinates

3. Specify texture parameters

- Wrapping - How should the text be wrapped around objects?
- Filtering

## Diagram

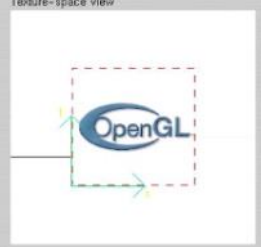
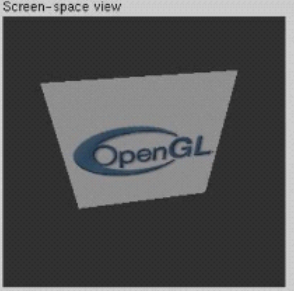


Geometric object - specified by  $x, y, z$  coordinates

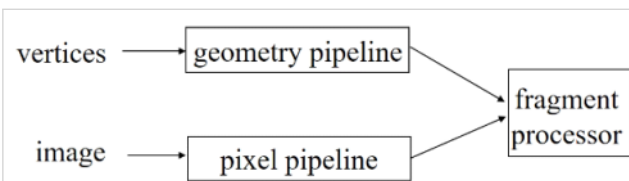
Flat image - specified by  $s$  and  $t$  coordinates

The geometric points are mapped onto the pixels of the image, producing the display

## Example

 <p>Texture-space view</p> <p>The image shows a square texture with a dashed red border. Inside the border is the OpenGL logo. A green arrow points from the left edge of the texture to the right edge, and a red arrow points from the top edge to the bottom edge, indicating the texture's dimensions.</p>	<p>This is a 256 x 256 image.</p> <p>It can be used as a texture map in OpenGL</p>
 <p>Screen-space view</p> <p>The image shows the OpenGL logo mapped onto a rectangular polygon in a perspective view. The logo is centered on the polygon, and the perspective effect makes the logo appear larger on the closer side of the rectangle.</p>	<p>The texture has been mapped to a rectangular polygon which is viewed in perspective</p> <p>The closer part looks bigger</p> <p>The texture size has changed accordingly</p>

## Mapping and Pipeline



The vertices go through the geometric pipeline (clipping, perspective project, interpolation), but the image goes through the pixel pipeline

Images and geometry flow through separate pipelines that *join during fragment processing*

Hence, "complex" textures do not affect geometric complexity

## Resolution



Only two triangles to form a simple square, but the texture makes it look detailed

We can have different resolutions for texture and geometry  
Texture processing is not as complex as geometry processing  
High resolution textures give more realistic appearance

**It is optimal to map High resolution textures onto Low resolution geometry**

This still looks good while being light on the graphics pipeline  
Ground textures in your Project are perfect examples of this

## Specifying a Texture Image

Define a texture image from an array of texels (texture elements) in CPU memory

```
// This is grayscale 512x512 texture (if it was colour there would be  
an extra dimension)
```

```
GLubyte my_texels [512] [512];
```

Define as any other pixel map

- Scanned image or camera image
- Generated by application program

Enable texture mapping

- `glEnable(GL_TEXTURE_2D)`
- OpenGL supports 1-4 dimensional texture maps
- We just use 2D textures in this unit



## Define Image as a Texture

```
glTexImage2D(target, level, components, w, h, border, format, type, texels);
```

1. **Target:** Type of texture (e.g. GL\_TEXTURE\_2D)
2. **Level:** The mipmapping level (e.g. 0)
3. **Components:** Elements per texel (e.g. 3)
4. **W, H:** Width and height of texels in pixels (e.g. 512, 512)
5. **Border:** Used for smoothing (e.g. 0)
6. **Format, Type:** Format and type of the texels (e.g. GL\_RGB, GL\_UNSIGNED\_BYTE)
7. **Texels:** Pointer to texel array (e.g. my\_texels)

## Typical Code

```
// Pass the vertex coordinates to vertex shader
offset = 0;
GLuint vPosition = glGetAttribLocation(shaderprogram, "vPosition");
glEnableVertexAttribArray(vPosition);
glVertexAttribPointer(vPosition, 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(offset));

// Piggy-back the texture coordinates at the end of the buffer and pass it
to vertex shader
offset += sizeof (points) ; // Update offset up to points
GLuint vTexCoord = glGetAttribLocation (program, "vTexCoord");
glEnableVertexAttribArray(vTexCoord);
glVertexAttribPointer(vTexCoord, 2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(offset));
```

# Texture Mapping 2

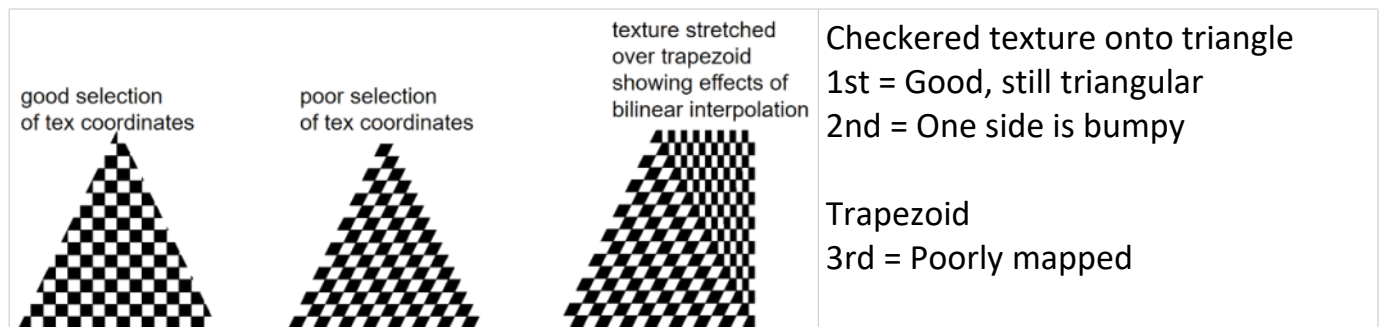
Wednesday, November 20, 2019 3:19 PM

## Interpolation

Defining texture coordinates for mapping textures onto rectangles is easy.

How to define texture coordinates for complex objects can be tricky.

OpenGL uses interpolation to find proper texels from specified texture coordinates. Distortions may result.



## Texture Parameters

OpenGL has a variety of parameters that determine how texture is applied

- **Wrapping parameters** determine what happens if  $s$  and  $t$  are outside the  $(0,1)$  range
- **Texture sampling** mode allows us to specify using area averaging instead of point samples. Recall that it is better to map from areas to areas rather than points to points
- **Mipmapping** allows us to use textures at multiple resolutions. We can use finer resolutions when are magnifying and lower resolutions when we are minimizing. It also helps to reduce aliasing.
- **Environment parameters** determine how texture mapping interacts with shading

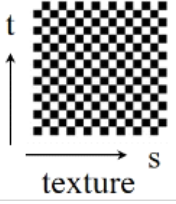

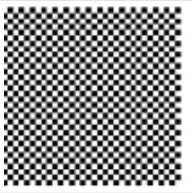
```
// X can be i or f
```

```
glTexParameterX(GLenum target, GLenum pname, GLint value);
```

```
// Example: S in 2nd param indicates we want to wrap in S dim. of s,t
```

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
```

## Wrapping Mode

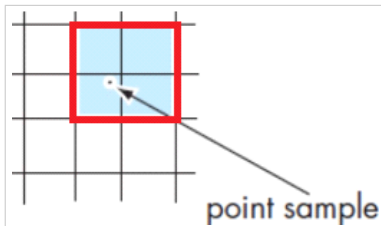
	<p><math>s</math> and <math>t</math> should generally be in the range 0 to 1. We can use clamping or wrapping to force them in the <math>[0,1]</math> range.</p>
	<p><b>Clamping:</b> Not very good <i>if <math>s</math> OR <math>t &gt; 1</math> use 1</i> <i>if <math>s</math> OR <math>t &lt; 0</math> use 0</i> <b>Code:</b> <code>glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);</code></p>
	<p><b>Wrapping:</b> Nicer looking, All four areas are the same <i>if <math>s, t &gt; 1</math>, then use modulo 1</i> <b>Code:</b> <code>glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);</code></p>

## Texture Sampling

Aliasing in textures is a major problem. When we map texture coordinates to the texels array, we rarely get a point that is exactly at the centre of the texel.

OpenGL supports the following options for sampling textures:

1. **Point sampling (Point to Point mapping)** — The value of the texel that is closest to the texture coordinates output by the rasterizer is used.
2. **Linear filtering (Area to Area mapping)** — The weighted average of a group of texels in neighborhood of the texture coordinates output by the rasterizer is used

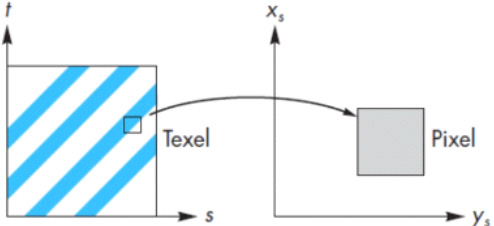
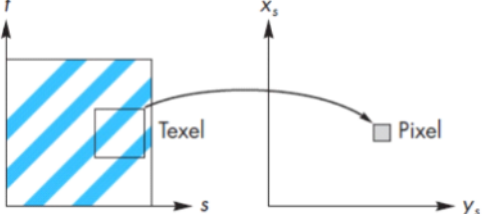


For point sampling, the value of the bottom-left cyan texel would be used.

For linear filtering, the weighted average of the four cyan texels would be used as the texture value for the point sample.

## Magnification and Minification

In deciding how to use the texel values to obtain the final texture value, the size of the pixel on the screen may be larger or smaller than the texel

	<b>Minification</b> One texel is smaller than one pixel Texel enlarges
	<b>Magnification</b> The texel is larger than one pixel Texel gets smaller



## OpenGL Code

The filter mode can be specified by calling:

```
glTexParameteri(target, type, mode);
```

### **Magnification:**

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

### **Minification:**

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

### **Notes:**

- For point sampling, replace GL\_LINEAR (Area to Area) by GL\_NEAREST (Point to Point)
- Linear filtering requires a border of an extra texel for filtering at edges (border = 1)

## Texture Arrays

For objects that project to a smaller area on the screen, we don't need to keep the original full resolution of the texel array.

OpenGL allows us to create a series of texture arrays at reduced sizes, e.g., for a 64 x 64 original array, we can set up 64 x 64, 32 x 32, 16 x 16, 8 x 8, 4 x 4, 2 x 2, and 1 x 1 arrays by calling:

```
glGenerateMipmap(GL_TEXTURE_2D);
```

The call generates low resolution textures that can be used for mag/minification, reducing aliasing effects

## Mipmapped Textures

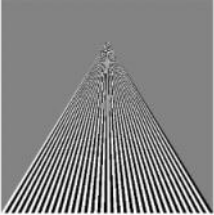
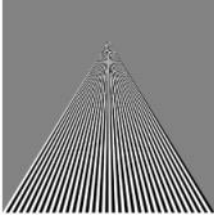
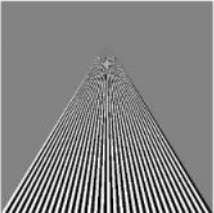

Mipmapping is a technique where an original high-resolution texture map is *scaled and filtered into multiple resolutions within the texture file*.

- Allows for prefiltered texture maps of decreasing resolutions
- Reduces interpolation errors for smaller textured objects

Thus the order of function calls is:

```
// You need to firstly declare the mipmap level during texture definition
// The second 0 indicates that we want to keep all resolutions (all the
// way to base level 0, i.e. the original texel array)
glTexImage2D( GL_TEXTURE_2D, 0 ... );
// Secondly, we generate the mipmap
glGenerateMipmap(GL_TEXTURE_2D);
```

## Examples

point sampling		linear filtering		Perspective view of jetty
mipmapped point sampling		mipmapped linear filtering		Point sampling = Distortion at far end Linear filtering = Less distortion at far end  Mipmapped: Point sampling = Better Linear filtering = Best result. No breaking down of texture at any scale, even at the very small scale

# Texture Mapping 3

Monday, 20 April 2020 7:23 PM

## Texture Functions

You can control how texture is applied by calling:

```
glTexEnv{fi} [v] (GL_TEXTURE_ENV, name, param);
```

**Variations:** Float, Integer, V is optional

- glTexEnvf
- glTexEnvfv
- glTexEnvi
- glTexEnviv
- ...

### name

- The symbolic name of a texture environment parameter
- e.g. GL\_TEXTURE\_ENV\_MODE

### param

- GL\_MODULATE: Modulate the texture with computed shade
- GL\_BLEND: Blend with an environmental color, Sets texture function to blend
- GL\_REPLACE: Use only texture color, Discard environment color

## Using Texture Objects

1. Specify textures in texture objects
2. Set texture filter
3. Set texture function
4. Set texture wrap mode (Wrap around or clamp)
5. Set optional perspective correction hint
6. Bind texture object
7. Enable texturing
8. Supply texture coordinates for vertex (coordinates can also be generated in vertex shader)

See example1.cpp in the CHAPTER07\_CODE folder

## Applying Textures

Textures are applied during fragment shading by a **sampler**

Samplers return a texture colour from a texture object

```
in vec4 color; //color from rasterizer
in vec2 texCoord; //texture coordinate from rasterizer
uniform sampler2D texture; //texture object from application

void main() {
    gl_FragColor = color * texture2D( texture, texCoord );
}
```

texture2D() is inbuilt GLSL function that takes the texture values and coordinate locations to produce a specific colour for each rasterized pixel

In variable comes from vertex shader/rast.  
texCoord is from rast.  
texture variable is from application

Fragment color = color X texture value



## Vertex Shader

Usually vertex shader will output texture coordinates to be rasterized

Must do all other standard tasks too:

- Compute vertex position
- Compute vertex color if needed

```
in vec4 vPosition; //vertex position in object coordinates
in vec4 vColor;    //vertex color from application
in vec2 vTexCoord; //texture coordinate from application

out vec4 color; //output color to be interpolated
out vec2 texCoord; //output texture coordinate to be
                  //interpolated
```

In variables are from application

Out variables are calculated inside vertex shader and passed down the pipeline, for interpolation

## Checkerboard Texture

We can create our own texture map in the application.

For example, creating a checkerboard texture:

```
GLubyte image[64][64][3];  
// Create a 64 x 64 checkerboard pattern  
for ( int i = 0; i < 64; i++ ) {  
    for ( int j = 0; j < 64; j++ ) { //bitwise &, Hex, ^ XOR  
        GLubyte c = (((i & 0x8) == 0) ^  
                     ((j & 0x8) == 0)) * 255;  
        image[i][j][0] = c;  
        image[i][j][1] = c;  
        image[i][j][2] = c;  
    }  
}
```

Partial code from [example1.cpp](#) in the [CHAPTER07\\_CODE](#) folder

## Adding Texture Coordinates

Code from an example in an earlier lecture:

```
void quad( int a, int b, int c, int d )
{
    quad_colors[Index] = colors[a];
    points[Index] = vertices[a];
    tex_coords[Index] = vec2( 0.0, 0.0 );
    index++;
    quad_colors[Index] = colors[a];
    points[Index] = vertices[b];
    tex_coords[Index] = vec2( 0.0, 1.0 );
    Index++;

    // other vertices
}
```

Partial code from `example1.cpp` in the `CHAPTER07_CODE` folder

Calculation of texture coordinates for vertices

## Texture Object

```
GLuint textures[1];
glGenTextures( 1, textures );

glBindTexture( GL_TEXTURE_2D, textures[0] );

glTexImage2D( GL_TEXTURE_2D, 0, GL_RGB, TextureSize,
              TextureSize, 0, GL_RGB, GL_UNSIGNED_BYTE, image );
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                GL_NEAREST );
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                 GL_NEAREST );
glActiveTexture( GL_TEXTURE0 );
```

Partial code from `example1.cpp` in the `CHAPTER07_CODE` folder

1. GLuint = Identifier for textures
2. Generate textures
3. Bind textures
4. Specify the texture mapping type
5. Set different parameters with `glTexParameterf()`
6. Activate texture

## Texture Mapping Real Images

If the texture to be mapped is a real image, it already has shading and shadows, as it comes from a real environment

- Due to directional light sources
- Due to self-occlusions and occlusions from other objects

These shading and shadows will not correspond to the light sources in your graphics world

A good texture map should **not include shading or shadows**

- Such an image can only be captured in controlled conditions with perfect diffused lighting i.e. light coming from all directions. This image is called an **albedo**
- Multiple large planar light sources can approximate diffused light

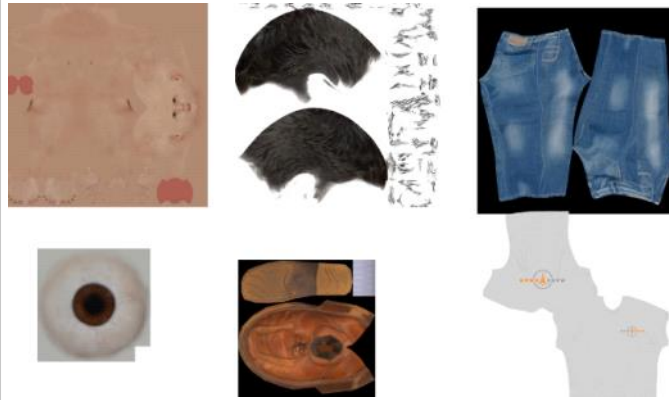
## 2D Texture on 3D Objects

Wrapping 2D texture onto a 3D object is problematic, especially if the 3D shape is complex, unless:

- The texture was on the 3D object already in the real world and the object was scanned
- Some effort was put into unwrapping the texture



## MakeHuman Texture Files



If we generate human with clothes option, we get texture files for the skin, hair, eyes, eyelashes, eyebrow, clothes, shoes, etc.

# Hierarchical Modelling

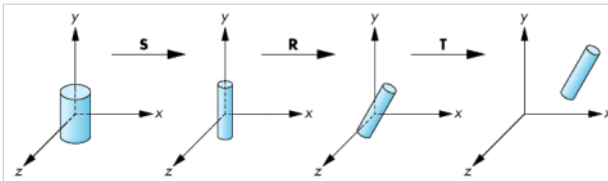
Hierarchical Modelling is a method of making complicated models from simple primitives



## Representing Complex Objects

A complex object can often be decomposed into **symbols** - simpler primitive parts

To render a complex object, we can transform these parts separately to fit the object. We must define an **instance transformation** for each part, e.g., a cylinder of radius 1 and height 1 is transformed to give a rod, which models an arm of a robot.



The instance can be represented by one transformation of the form  $M = TRS$  Scaling, then Rotation, then Translation. Rotations needs to happen while still at the origin

```
mat4 instance;  
mat4 model_view;  
instance = Translate(dx, dy, dz)*RotateZ(rz)* RotateY(ry)*RotateX(rx)*Scale(sx, sy, sz);  
model_view = model_view*instance;  
cylinder(); /* or some other symbol */
```


Two matrices: Instance transformation and model view

Instance transformation is calculated by applying the transformations

## Symbol-Instance Table

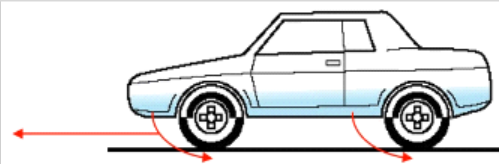
For the example before, we would probably have **8 different instances** of the cylinder to represent the left/right upper/lower arms and legs of the robot.

We can store each symbol instance into a symbol table for the complex object by assigning a number to each symbol instance and storing the parameters for the instance transformation.

2 <sup>nd</sup> and 3 <sup>rd</sup> instances of symbol 1	Symbol	Scale	Rotate	Translate	1 represents the cylinder which is being re-used	
	1	$s_x, s_y, s_z$	$\theta_x, \theta_y, \theta_z$	$d_x, d_y, d_z$		
	2					
	3					
	1					
	1				8 cylinders form limbs	
	.					
	.					
	.					
	.					

## Relationship of Parts in a Car Model

The problem with a **symbol-instance table** is it does not show relationships between different parts of the complex model



Consider the modelling of a car:

- Chassis + 4 identical wheels
- Two symbols

Rate of forward motion determined by rotational speed of wheels - cannot be represented with the symbol-instance table

## Pseudocode

```
float s; /* speed */
float d[3]; /* direction */
float t; /* time */
/* determine speed and direction at time t */
draw_right_front_wheel(s,d);
draw_left_front_wheel(s,d);
draw_right_rear_wheel(s,d);
draw_left_rear_wheel(s,d);
draw_chassis(s,d);
```

In pseudocode, our rendering of the car

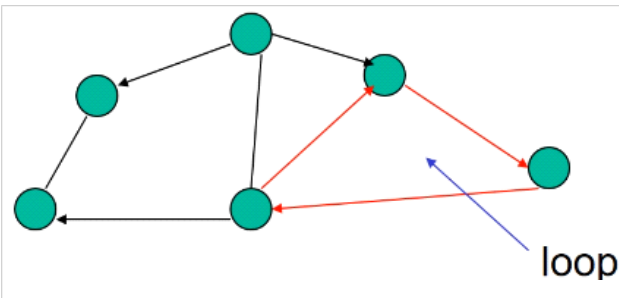
Direction is a vector

Each part is drawn separately

It fails to show the relationships of the different parts of the car well (e.g., the wheels do not rotate independently; the chassis must move together with the wheels).

## Graphs

We could **use a graph** to better represent the relationship between the parts of the car

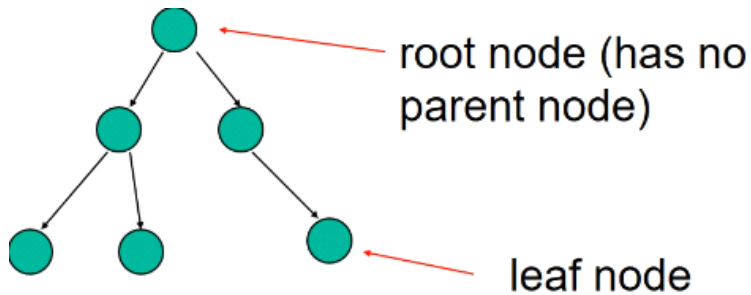


- A graph consists of a number of **nodes** and **edges/links**
- An edge connects a pair of nodes (Edges can be *directed or undirected*)
- A **cycle graph** is a directed path that has at least one loop

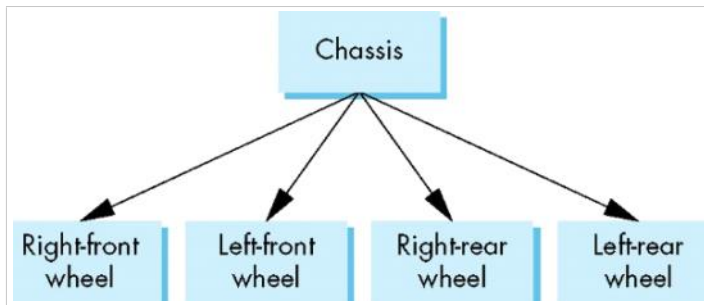
## Trees

A **tree** is a graph in which each node (except the root) has **exactly one parent node** (This implicitly imposes that the graph cannot have loops) but:

- May have multiple children
- May have no children (such nodes are known as leaf or terminal nodes)

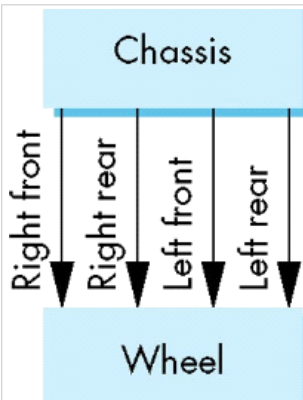


## Tree Model of Car



Chassis is the root node,  
and has four children

## DAG Model



We can also use a **directed acyclic graph (DAG)** to represent the car

This uses the fact that all the wheels are identical

We have one wheel with four connections

This is not much different than dealing with a tree



### **Graphs** are useful for:

- Moving the complete object
- Animating parts of the object

# Hierarchical Modelling with Trees

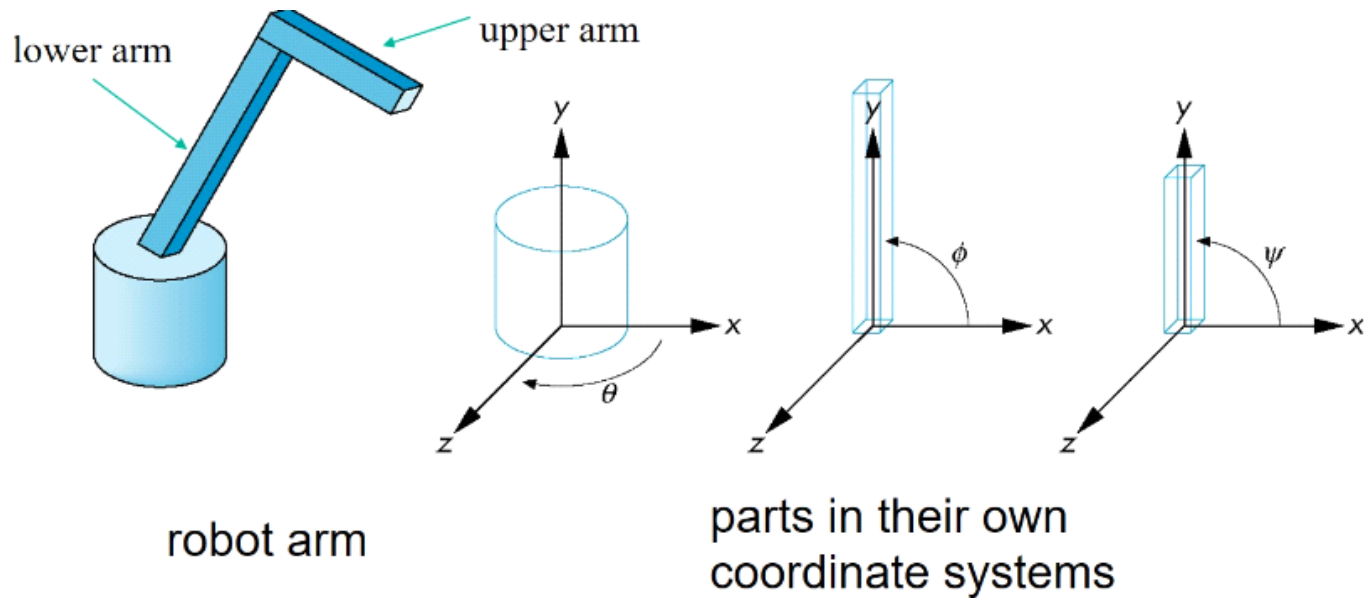
Thursday, 23 April 2020 1:37 PM

## Modelling with Trees

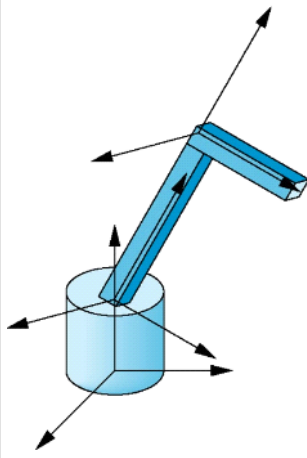
To model our complex object using a tree structure, we:

- Must decide what information to place in the nodes and what to put in the edges
- For nodes, define
  - What to draw, and
  - Pointers to all the child nodes (to maintain tree structures)
- For edges, we may have information on incremental changes to transformation matrices (which can also be stored in the nodes)

## Example: Robot Arm



## Articulated Models



The robot arm is an example of an **articulated model** where

- Adjacent parts are connected at a joint
- We can specify an instance appearance of the articulated model by giving all joint angles

## Relationships in Robot Arm

### **Base rotates independently**

- Single angle determines position

### **Lower arm attached to base**

- Its position depends on rotation of base - it must move along with base
- Must also translate relative to base and rotate about connecting joint

### **Upper arm attached to lower arm**

- Its position depends on both base and lower arm
- Must translate relative to lower arm and rotate about joint connecting to lower arm

## Required Matrices

Rotation of base: $R_b$ - Apply $M = R_b$ to base	Least transformations as closest to root
Translate lower arm <u>relative</u> to base: $T_{lb}$	
Rotate lower arm around joint: $R_{la}$ - Apply $M = R_{la} T_{lb} R_b$ to lower arm	Rotation of base -> Translation of lower base -> Rotation of lower arm
Translate upper arm <u>relative</u> to lower arm: $T_{ul}$	
Rotate upper arm around joint: $R_{ua}$ - Apply $M = R_{ua} T_{ul} R_{la} T_{lb} R_b$ to upper arm	Add: Translation of upper limb, Rotation of upper arm Most transformations as far from root

## OpenGL Code for Robot

```
mat4 modelMatrix;
```

Rotation applies to Base

```
void robot_arm()
```

Lower arm gets three matrices

```
{
```

```
    modelMatrix = RotateY(theta);
```

```
    base();
```

Upper arm get 5 matrices

```
    modelMatrix *= Translate(0.0, h1, 0.0);
```

```
    modelMatrix *= RotateZ(phi);
```

```
    lower_arm();
```

```
    modelMatrix *= Translate(0.0, h2, 0.0);
```

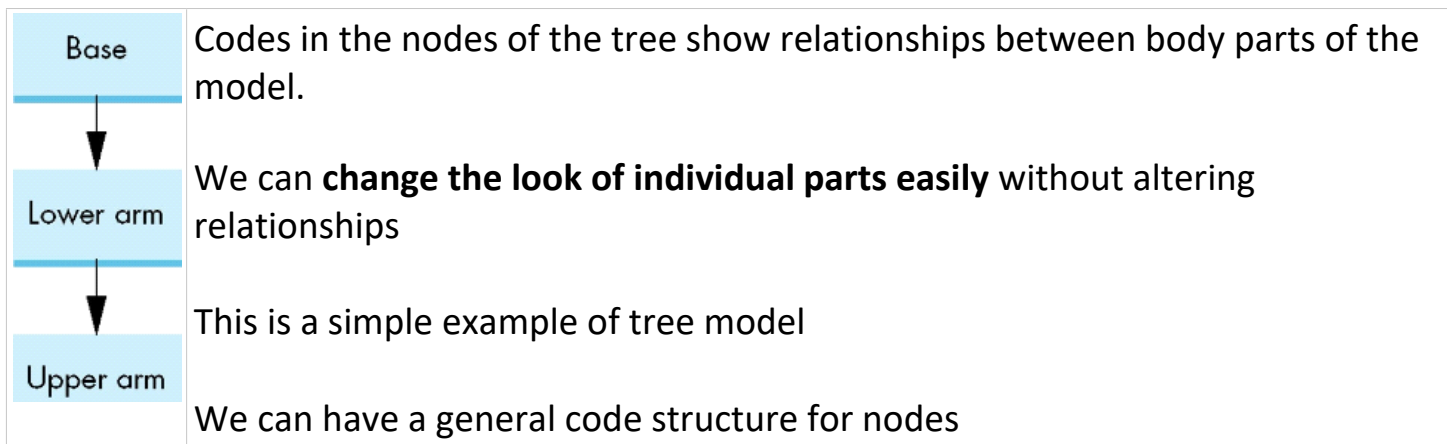
```
    modelMatrix *= RotateZ(psi);
```

```
    upper_arm();
```

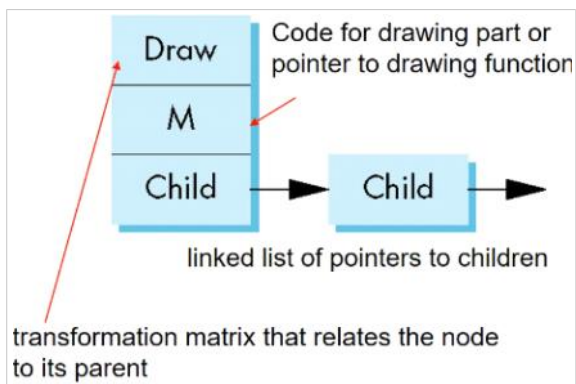
```
}
```



## Tree Model of Robot



## Possible Code Structure



If we store all the necessary information in the nodes, rather than in the edges, then **each node must store at least three items:**

1. Draw = A pointer to a function that draws the object represented by the node
2. M = A homogeneous-coordinate matrix that positions, scales, and orients this node (and its children) relative to the node's parent
3. Child = Pointers to children of the node

# Generalizing Hierarchical Modelling

Thursday, 23 April 2020 1:37 PM

## Generalizations

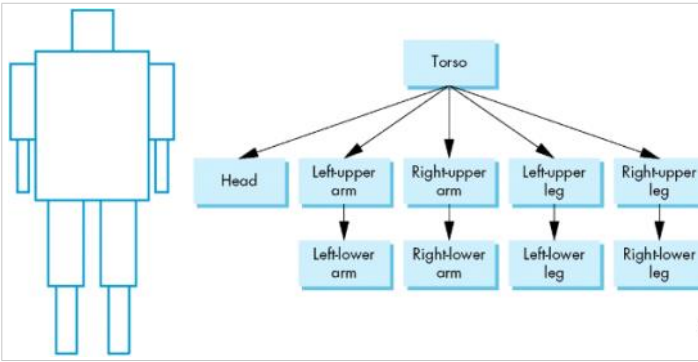
### **Need to deal with multiple children**

- How do we represent a more general tree?
- How do we traverse such a data structure?

### **Animation**

- How to use dynamically?
- Can we create and delete nodes during execution?

## Humanoid Figure



The humanoid robot has **10 body parts**:

1. Torso (Root node, as is connected to most elements)
2. Head
3. Left-upper and left-lower arms
4. Right-upper and right-lower arms
5. Left-upper and left-lower legs
6. Right-upper and right-lower legs.

## Building the Model

- Can build a simple implementation using quadrics: ellipsoids and cylinders (to represent the body parts)
- Access body parts through function calls:
  - `torso ( )`;
  - `left_upper_arm ( )`;
- The transformation matrix stored in a node describes the position of the node with respect to its parent
  - Example:  $M_{lla}$  positions left lower arm with respect to left upper arm

## Display and Traversal

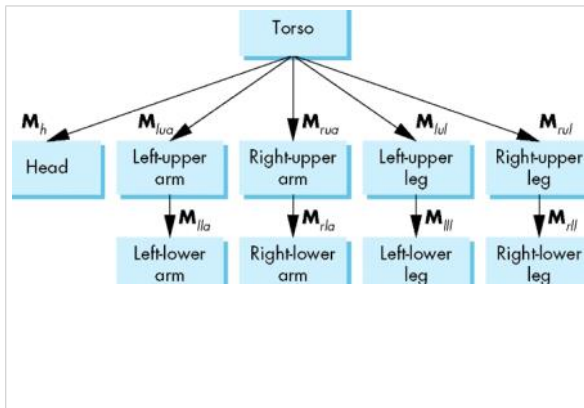
The position of the robot is determined by **the joint angles** or the transformation matrices of each part

Display of the tree requires a **tree traversal** (depth or width first?)

- Usually we go from left to right, going all the way down to leaf node each time
- Visit each node once.
- Call the display function at each node that describes the part associated with the node
- We do two transformations = 1st to create part, 2nd to animate

This involves applying the correct transformation matrix for the position and orientation of the part.

## Transformation Matrices



There are 10 relevant matrices

- $M$  positions and orients entire figure through the torso which is the root node
- $M_h$  positions head with respect to torso
- $M_{lua}, M_{rua}, M_{lul}, M_{rul}$  position upper arms and legs with respect to torso
- $M_{lla}, M_{rla}, M_{lll}, M_{rll}$  position lower parts of limbs with respect to corresponding upper limbs

For more natural (less robotic) figures, **skinning** is a common animation technique that uses weighted averages of transformations for vertices near joints (more realistic)