19 Interactive Three-dimensional Graphics
Shapes, Lights, Textures, Transforms
Overview

- In this lecture we discuss how to achieve interactive 3d programming in processing
  - We revisit a number of previous issues e.g. modelling, light, camera modelling
  - However we approach things in a procedural way:

  We don’t just describe a scene/image geometrically BUT a set of steps that achieves the creation of a scene

  Everything we create will be potentially interactive
Topics Covered

- 3D object creation
- Lighting
- Texturing
- Transformations and Viewing
- Loading 3D objects
Interactive 3D Graphics: Tools such as 3D Studio Max

Each of the editing windows is real-time Interactive 3D Graphics. As opposed to the rendered production output (non-realtime)
Interactive 3D Graphics: Games

Quake III Arena - Id Games 2000
Graphics Acceleration

- Specialised hardware for graphics operations
  - Transformations
  - Viewing
  - Raster operations
  - Texture operations

- Pipelined parallel hardware techniques which speeds up processing of stream data (commonly used in graphics)
- Frees up CPU cycles for other tasks
3D in processing

- 3D capabilities in processing include:
  - 3D coordinates
    - Points have x, y, and z values
  - 3D primitives
    - beginShape()
  - Camera
    - Positioning
    - Projection
  - Lights + Illumination
  - Texture mapping
- Pshape class allows loading of 3D objects e.g. from 3D Studio Max
Setting up 3D

- In the size call, add a third parameter: OpenGL or P3D
  - Instead of: `size(400, 400);`
  - Use: `size(400, 400, P3D);`

- Note that processing’s coordinate system has the origin (0,0,0) at the top left
  Positive Z comes out of the screen (left hand rule)
Some 3D objects

- 2D Objects work equally well
  - ellipse, rect, line
  - These can be transformed in 3D
  - N.B. A few 2D functions are not valid in 3D e.g. smooth may no longer work

- Processing is designed to seamlessly merge 2D and 3D aspects

- Additionally 3D objects are provided
  - box, sphere,
  - generic primitives through beginShape

- There are some functions that are uniquely 3D and don’t work at all in 2D mode

If in doubt, always check the reference manual (in fact, you should always read the entry for every function you plan to use): [http://processing.org/reference/](http://processing.org/reference/)
The `box` function sets the size of the box. If only one value is given, `width`, `height`, and `depth` are all set to this value.

Note that only the size is set within the `box` function. Positioning and orientation are done through transforms: `rotate`, `translate`.
A sphere is made up of lots of triangles (these are created for you by processing)

Advanced: You can change the **Level of Detail** using `sphereDetail(..)`

- `sphereDetail(20)`
- `sphereDetail(10)`
- `sphereDetail(6)`
- `sphereDetail(4)`
Polygon Mesh

- **Essentially:**
  - Vertices
  - Polygons
- **Commonly Triangle Meshes**
- **Optionally:**
  - Normals
  - Colours
  - Texture Co-ordinates

In modern day applications it is common to have:
- 10,000+ polygons per object
- 10,000,000+ polygons in a scene
Other shapes

- The following combination of functions allows you to create your own shapes
  - beginShape(...);
  - vertex(...);
  - ...
  - endShape();

- Vertex defines the points that makes up the shape (by default connect up all dots and fill in)
  - Simple Example
    - beginShape();
    - vertex(30, 20);
    - vertex(85, 20);
    - vertex(85, 75);
    - vertex(30, 75);
    - endShape();

Usually we use many calls to vertex
3D Primitives

The parameter we provide to `beginShape` affects how the vertices are connected up.

```
beginShape(POINTS);
vertex(30, 20);
vertex(85, 20);
vertex(85, 75);
vertex(30, 75);
endShape();
```

```
beginShape(LINES);
vertex(30, 20);
vertex(85, 20);
vertex(85, 75);
vertex(30, 75);
endShape();
```

```
noFill();
beginShape();
vertex(30, 20);
vertex(85, 20);
vertex(85, 75);
vertex(30, 75);
endShape();
```

```
noFill();
beginShape();
vertex(30, 20);
vertex(85, 20);
vertex(85, 75);
vertex(30, 75);
endShape(CLOSE);
```

Note that otherwise the code here is unchanged from the previous slide.
3D Primitives

Every 3 vertices grouped together to form a triangle.

First 3 vertices grouped together to form a triangle. Then the last two from previous triangle are grouped with next vertex to form the next triangle etc...
3D Primitives

Please see documentation online for further details
PShape

- For complex shapes you obviously don’t want to manually type every single vertex in
- Processing provides a built in Class PShape that allows you to load and manage 3D objects
- Can use this, for instance, to load 3D objects made in a tool (such as 3DS Max)
- In 3DS Max:
  - select an object.
  - Click on File Export
  - Export to OBJ format

```java
PShape s;

void setup()
{
    s = loadShape("virtual-trinity-old.obj");
}

void draw()
{
    background(0);
    translate(width/2, height/2);
    scale(1, -1, 1);
    shape(s, 0, 0);
}
```
Transformations

- Transformations functions in processing include:
  - translate \((tx, ty, tz)\)
  - scale \((sx, sy, sz)\)
  - rotate \((\text{radians}, ax, ay, az)\)

- Easier options for rotation:
  - rotateX( radians );
  - rotateY( radians );
  - rotateZ( radians );

- Each of these affect the “Current Transformation”
  - IMPORTANT NOTE: a transform function affects all objects drawn AFTER it
    - Think of it as moving the pen to the place you want to draw
  - Transformations are cumulative (can combine several operations by calling them one after the other)

**NOTE!:** These also apply to lights, camera, images as well as predefined shapes

- radians is a unit of angle (similar to degrees but bigger)
- You can convert from degrees to radians using the `radians( )` function
Using Transforms

- In the program, we first call a transformation function and then draw/create an object (e.g. box, sphere, or beginShape)

```java
int angle = 0;

void setup()
{
    size (400, 400, P3D);
}

void draw()
{
    background(0);
    noFill();
    stroke(255);
    translate(200, 200, 200);
    rotate( radians( angle ), 0, 1, 0);
    sphere(20);
    box (50);
    angle++;
}
```

These transforms apply to all objects drawn afterwards.
Advanced: Cumulative Transforms

- Normally objects are affected by all previous transforms that have gone before:

  ```
  translate (0, 10, 0);
  sphere(20); // this is translated by <0, 10, 0>
  translate (100, 0, 0);
  sphere(10); // this is translated by <0, 10, 0> AND <100, 0, 0>
  ```
Advanced: Dependent Transforms

- By default this means all objects are dependent on each other
  - This is good for **hierarchical objects**

- Bad For independent objects unless we have some way of separating them

```cpp
int angle = 0;
void draw()
{
    background(0);
    lights();
    fill(255);
    noStroke();
    translate(200, 200, 200);

    rotate(3*cos( radians( angle )), 0, 0, 1);
    translate(0, 10, 0);
    box (5, 20, 5);

    translate(0,10, 0);
    sphere(4);
    rotate( sin(radians( angle )), 0, 0, 1);
    translate(0, 10, 0);
    box (5, 20, 5);

    translate(0,10, 0);
    sphere(4);
    rotate( sin(radians( angle )), 0, 0, 1);
    translate(0, 10, 0);
    box (5, 20, 5);

    angle++;
}
```
Advanced: Independent transforms

- `resetMatrix()` tells the system to forget all previous transforms that have been applied.

  ```
  translate (0, 10, 0);
  sphere( 20);  // this is translated by <0, 10, 0>
  
  resetMatrix();
  translate (100, 0, 0);
  sphere( 10);  // this is translated by <100, 0, 0>
  ```
Advanced: Hierarchical Transforms

- `pushMatrix()` and `popMatrix()` allow you to save and retrieve transforms from a transform Stack

```
translate (0, 10, 0);
sphere (20);  // this is translated by <0, 10, 0>
pushMatrix();

rotate ( 45, 0, 0, 1);  // rotate by 45 about z axis
box (30);  // this is translated by <0, 10, 0> AND rotated by 45 about z axis
popMatrix();  // retrieves CTM before the rotate

translate (100, 0, 0);
box( 20);  // this is translated by <0, 10, 0> AND <100, 0, 0>
```

- This can be used for modelling some interesting hierarchical objects
Camera

- Use the `camera(…)` function to align and configure the camera
- Sets the position of the camera based on
  - the eye position,
  - The target point OR the centre of the scene OR point of “focus”
  - and which axis is facing upward.
- Moving the eye position and target allows the images to be seen from different angles.
Camera Position

- camera (ex, ey, ez, tx, ty, tz, ux, uy, uz);

<ex, ey, ez> is a vector defining the **camera position** (a.k.a. eye position)

<tx, ty, tz> is a vector defining the **target point** (the point the camera is looking at)

<ux, uy, yz> is the **up vector** which loosely orients the camera. Make sure this is not parallel to the camera direction.
Camera example

```
size(100, 100, P3D);
noFill();
background(204);
camera( 70.0, 35.0, 120.0,
    50.0, 50.0, 0.0,
    0.0, 1.0, 0.0);
translate(50, 50, 0);
rotateX(-PI/6);
rotateY(PI/3);
box(45);
```
Projection

Perspective projections

```cpp
noFill();

float fov = PI/3.0;

float cameraZ = (height/2.0) / tan(PI * fov / 360.0);

perspective(fov, 
    float(width)/float(height),
    cameraZ/10.0, cameraZ*10.0);

translate(50, 50, 0);
rotatex(-PI/6);
```

Orthographic projection

```cpp
noFill();

ortho(0, width, 0, height, -10, 10);
translate(100, 100, 0);
rotatex(-PI/6);
rotatey(PI/3); box(45);
```

Perspective is the default and is already set up. You don't really have to do any of this yourself unless you want to change these parameters.
Lighting

- **In interactive applications:**
  - Usually only local illumination models
  - Texture mapping – multi-texturing

- **In some applications**
  - Shadow mapping
  - Bump mapping
  - Environment mapping
Lightsource

- Use `pointLight(...)` to place a light source in the scene
  - `pointLight(v1, v2, v3, x, y, z)`
    - First three values are RGB colour of light
    - x, y, z define the position of the light source

- Other Light functions
  - `directionalLight()`
  - `lightFalloff()`
  - `lightSpecular()`
  - `ambientLight()`
  - `spotLight()`
  - `lights`
Types of Lighting in Processing

directionalLight(255, 255, 255, 0, -10, -20);

pointLight(255, 255, 255, 0, 100, 400);

// Slightly more realistic

// light varies across a polygon

ambientLight(51, 102, 126);

// sets ambient lighting in the scene
Lights

- Lights can be placed in the scene using either the pointLight or directionalLight functions:
  - `pointLight( r, g, b, px, py, pz );`
  - `directionalLight( r, g, b, dx, dy, dz );`

  <r, g, b> is the colour of the light
  <px, py, pz> is the position of a point light in 3D space
  <dx, dy, dz> is a directional vector for light rays from a distant source (see 3DS Max Notes)

- e.g. `pointLight( 255, 255, 255, 0, 100, 100 );`
  A white point lightsource at position: <0, 100, 100>
pointLight Example

```
size(100, 100, P3D);
background(0);
noStroke();
pointLight(51, 102, 126, 35, 40, 36);
translate(80, 50, 0);
sphere(30);
```

Material Properties

- Controlled using:
  - shininess()
  - specular()
  - ambient()
  - emissive()

- Fill color sets the main (diffuse colour)

- Call these functions before drawing the object

- Generally applies to everything following it
Texture Mapping in Processing

- Given:
  - A 3D model (defined as a set of vertices)
  - A texture defined as an image
- We need to associate each vertex of the model with a point on the texture

```java
PImage p;
p = loadImage( "wood8.jpg");

beginShape(TRIANGLES);
texture(p);
vertex (0, 0, 100 , 142, 3);
vertex (94, 0, -33.3, 0, 221);
vertex (-47, 81, -33.3, 264, 216);
endShape();
```
Texture mapping

- An image texture can be applied to an object defined with `beginShape`
- Apart from the \(<x, y, z>\) co-ordinates of the vertex, you need to provide the UV coordinates on the image where the colour of the pixel comes from.
Texture mapping Example

```javascript
noStroke();

PImage a = loadImage("face.jpg");

beginShape(QUADS);

texture(a);

vertex(0,0, 25, 32);

vertex(200, 0, 0, 75, 31);

vertex(200, 200, 0, 65, 80);

vertex(0, 200, 0, 30, 81);

endShape();
```

N.B. The `texture()` function must be called between `beginShape()` and `endShape()` and before any calls to `vertex()`.
Lab 19
Processing and 3D
Objectives for this lab

- **Create a 3D scene in processing**
  - Start Processing in 3D Mode (P3D)
  - Set the background colour to green <0, 0, 255>
  - Create a 3D shape. Either one of
    - sphere
    - cube
    - arbitrary shape (using beginShape)
  - Add lights to the scene
  - Add a texture to the shape
  - Use mouse or keyboard input to move/rotate objects OR the camera view

- **Advanced Options (do at least one of the following):**
  - Use the built in PShape object to load an OBJ file (export from 3DS max or downloaded from web)
  - Install and use the **PeasyCam** library to rotate and move the scene
Suggested Scene

Try to draw this scene with a sphere resting on a box (don’t worry if you don’t get it exactly identical). Use transforms or camera functions to move the view of the scene interactively.
Suggested Scene: MORE ADVANCED

- Rotating textured box
  - Interaction:
    - Hit w/s to rotate up / down
    - Hit a/d to rotate left right

N.B. These are just the suggested positions of the vertices provided for your convenience. You need to call `beginShape` and make QUADS out of these to make the box, you also need to assign the appropriate texture coordinates (two additional values in the `vertex` function NOT SHOWN HERE).
Suggested Scene: MORE ADVANCED

- Rotating textured tetrahedron
- Different texture on each face

- Tetrahedron vertices
  
  vertex(0, 100, 0);  //top
  vertex (94, -33.3, 0);  //bottom-left
  vertex(-47, -33.3, 81);  //bottom-right-far
  vertex(-47, -33.3, -81);  //bottom-right-near

N.B. These are just the suggested positions of the vertices provided for your convenience. You need to call `beginShape` and make `TRIANGLES` out of these to make the tetrahedron, you also need to assign the appropriate texture coordinates (two additional values in the `vertex` function NOT SHOWN HERE).
Setting Up for 3D

- Similar to a normal 2D sketch except when we call `size()` we give it an additional parameter: `P3D`

- It is useful (though not compulsory) to move to the centre of the screen before doing 3D operations
Creating Arbitrary Shapes

- This is typically done using a combination of 3 functions:
  - `beginShape(...)`
  - `vertex( ... )`
  - `endShape()`

```java
void draw()
{
    background(0);
    translate(width/2, height/2);
    beginShape();
    vertex(-100, -100, 0);
    vertex(-100, 100, 0);
    vertex(100, 100, 0);
    vertex(100, -100, 0);
    endShape();
}
```
Transforming Shapes

- Available functions
  - `translate( ... );` // a.k.a. move
  - `rotate ( ... );`
  - `scale ( ... );` // a.k.a. resize

- Typically you call this before defining the shape
  - In fact it applies to everything after the call

- You can call multiple transforms

```java
void draw()
{
    background(0);
    translate(width/2, height/2);
    scale(0.5, 0.5, 0.5);  // halve size
    rotateX(radians(90));  // rotate 90 degrees
    beginShape();
    vertex(-100, -100, 0);
    vertex(-100, 100, 0);
    vertex(100, 100, 0);
    vertex(100, -100, 0);
    endShape();
}
```
### Texture

- Declare a texture image using the `texture(…)` function
- Then you need to provide additional parameters to the `vertex(…)` calls
  - Texture coordinates indicate for each vertex, the coordinates on the texture image from which to get its colour
  - In-between values are then interpolated

```cpp
PImage img;
void setup()
{
    img = loadImage("texture.jpg")
}

void draw()
{
    background(0);
    translate(width/2, height/2);
    beginShape();
    texture( image );
    vertex(-100, -100, 0, 20, 20 );
    vertex(-100, 100, 0, 20, 200 );
    vertex( 100, 100, 0, 180, 200);
    vertex( 100, -100, 0, 180, 20);
    endShape();
}
```
Built-in 3D Shapes

- The following provided by Processing. Essentially these will replace the need for `beginShape()` and `vertex()`.
  - `sphere(...);`
  - `box(...);`

- Note that some 2D shapes also work in 3D. E.g.:
  - `ellipse(...)`
  - `line(...)`
  - `rect(...)`

```java
void draw()
{
    background(0);
    translate(width/2, height/2);
    sphere(40); //create sphere of radius 40
    translate(0, 50, 0);
    box(100, 20, 100);
}
```
Lights

- `lights()` will set up a Lightsource with default values

- More specific types of lights are created using one of the following functions:
  - `pointLight()`
  - `directionalLight()`
  - `spotLight()`
  - `ambientLight()`

- Material properties are set using:
  - `fill( ... );` //sets the diffuse colour
  - `specular(...);`
  - `shininess(...);`
  - `ambient ( ... );`
  - `emmisive (...);`

```java
void draw()
{
  background(0);
  translate(width/2, height/2);
  pointLight(255, 255, 255,
             400, -400, -400);
  ambientLight(40, 40, 40);
  noStroke();
  fill(200, 100, 100);
  sphere(40);
}
```
Camera

- Camera type is set using one of the following:
  - perspective(...);
    - Perspective camera.
    - Set by default
  - ortho(...)
    - Orthographic camera
- The camera() function lets you place and point the camera in a specific direction
  - Easier option is to install the PeasyCam library

```cpp
void draw()
{
  background(0);
  ortho();
  camera(0, 0, -100, 0, 0, 0);
  translate(width/2, height/2);
  box(100, 100, 100);
}
```

```cpp
void draw()
{
  background(0);
  perspective();
  translate(width/2, height/2);
  box(100, 100, 100);
}
```
Loading 3D Objects: PShape

- Processing provides a built in Class PShapes that allows you to load and manage 3D objects
- We can use this, for instance, to load 3D objects made in a tool (such as 3DS Max)
- In 3DS Max:
  - select an object.
  - Click on File Export
  - Export toOBJ format
- Other examples in next slide

```java
PShape s;

void setup()
{
  s = loadShape("virtual-trinity-old.obj");
}

void draw()
{
  background(0);
  translate(width/2, height/2);
  scale(1, -1, 1);
  shape(s, 0, 0);
}
```
OBJ Files

- Online examples (see next page):

- Get others here (look for free OBJ files)
  - http://tf3dm.com/3d-models/architecture
Camera Library

- **PeasyCam** is a popular third-party library
- **Simplifies movement of the camera**

**TO INSTALL:**
- Go to Sketch → Import Library → Add Library
- Search for PeasyCam and install it
- Go to Sketch → Import Library → PeasyCam

- Just add a few lines to setup and it should automatically rotate your scene when you drag the mouse

```java
import peasy.*;
import peasy.org.apache.commons.math.*;
import peasy.org.apache.commons.math.geometry.*;

PShape s;
PeasyCam cam;

void setup() {
    size(800, 800, P3D);
    cam = new PeasyCam(this, 100);
    cam.setMinimumDistance(200);
    cam.setMaximumDistance(500);

    // The "obj" file must be in the data folder of the current sketch to load successfully
    s = loadShape("virtual-trinity-old.obj");
}

void draw() {
    background(100, 130, 240);
    scale(1, -1, 1);
    shape(s, 0, 0);
}
```