

OpenGL

---

---

---

---

---

---

---

---

- ### OpenGL
- OpenGL
    - Is a mechanism to create images in a frame buffer
    - Is an API to access that mechanism
    - Is well specified
  - OpenGL
    - Is not a window system
    - Is not a user interface
    - Is not a display mechanism
    - Does not even own the framebuffer
      - It is owned by the window system so it can be shared
      - But OpenGL defines its attributes carefully

---

---

---

---

---

---

---

---

### White-Square Code

```

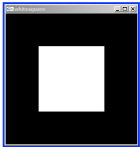
// Draw a white square against a black background
#include <windows.h>
#include <stdio.h>
#define GLUT_DISABLE_ATEXIT_HACK // yuck!
#include <GL/glut.h>

void draw() {
    glClear(GL_COLOR_BUFFER_BIT);
    glLoadIdentity();
    glOrtho(0, 4, 0, 4, -1, 1);
    glBegin(GL_POLYGON);
    glVertex2i(1, 1);
    glVertex2i(3, 1);
    glVertex2i(3, 3);
    glVertex2i(1, 3);
    glEnd();
    glFlush();
}

int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
    glutCreateWindow("WhiteSquare");
    glutDisplayFunc(draw);
    glutMainLoop();
}
    
```

OpenGL {

GLUT {




---

---

---

---

---

---

---

---

### OpenGL Portion of White-Square Code

```

glClear(GL_COLOR_BUFFER_BIT); // black background
glLoadIdentity();
glOrtho(0, 4, 0, 4, -1, 1); // int cast to double
glBegin(GL_POLYGON); // draw white square
    glVertex2i(1, 1);
    glVertex2i(3, 1);
    glVertex2i(3, 3);
    glVertex2i(1, 3);
glEnd();
glFlush(); // force completion
    
```

---

---

---

---

---

---

---

---

### Red-Book Example

```

glClearColor(0.0, 0.0, 0.0, 0.0);
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(1.0, 1.0, 1.0);
glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
glBegin(GL_POLYGON);
    glVertex3f(0.25, 0.25, 0.0);
    glVertex3f(0.75, 0.25, 0.0);
    glVertex3f(0.75, 0.75, 0.0);
    glVertex3f(0.25, 0.75, 0.0);
glEnd();
glFlush(); // force completion
    
```

---

---

---

---

---

---

---

---

### State tables

COLOR\_CLEAR\_VALUE 0,0,0,0

Get value	Type	Get Cmd	Initial Value	Description	Sec.	Attribute
DRAW_BUFFER	$1 + \times Z_{16}$	GetIntegerv	see 4.2.1	Draw buffer selected for output color i	4.2.1	color-buffer
INDEX_WRITEMASK	$Z^+$	GetIntegerv	1%	Color index writemask	4.2.2	color-buffer
COLOR_WRITEMASK	$4 \times B$	GetBooleanv	True	Color write enables R, G, B, or A	4.2.2	color-buffer
DEPTH_WRITEMASK	$B$	GetBooleanv	True	Depth buffer enabled for writing	4.2.2	depth-buffer
STENCIL_WRITEMASK	$Z^+$	GetIntegerv	1%	Front stencil buffer writemask	4.2.2	stencil-buffer
FRONT_STENCIL_WRITEMASK	$Z^+$	GetIntegerv	1%	Back stencil buffer writemask	4.2.2	stencil-buffer
COLOR_CLEAR_VALUE	$C^4$	GetFloatv	0.0, 0.0, 0.0, 0.0	Color buffer clear value (RGBA mode)	4.2.3	color-buffer
INDEX_CLEAR_VALUE	$C^1$	GetFloatv	0	Color buffer clear value (color index mode)	4.2.3	color-buffer
DEPTH_CLEAR_VALUE	$I^+$	GetIntegerv	1	Depth buffer clear value	4.2.3	depth-buffer
STENCIL_CLEAR_VALUE	$Z^+$	GetIntegerv	0	Stencil clear value	4.2.3	stencil-buffer
ACCUM_CLEAR_VALUE	$4 \times I^+$	GetFloatv	0	Accumulation buffer clear value	4.2.3	accumu-buffer

OpenGL 2.0 Spec, Table 6.21. Framebuffer Control

---

---

---

---

---

---

---

---



## The OpenGL Pipeline

---

---

---

---

---

---

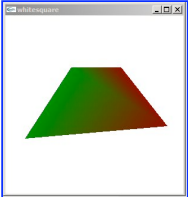
---

---

### OpenGL Shaded-Quad Code

```

glClearColor(1, 1, 1, 1); // white
glClear(GL_COLOR_BUFFER_BIT);
glLoadIdentity();
glOrtho(0, 100, 0, 100, -1, 1);
glBegin(GL_TRIANGLE_STRIP);
    glColor3f(0, 0.5, 0); // dark green
    glVertex2i(11, 31);
    glVertex2i(37, 71);
    glColor3f(0.5, 0, 0); // dark red
    glVertex2i(91, 38);
    glVertex2i(65, 71);
glEnd();
glFlush();
    
```




---

---

---

---

---

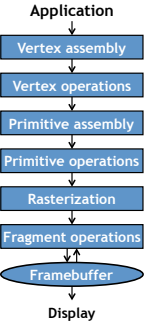
---

---

---

### OpenGL Pipeline

- Emphasis is on data types
- Diagram ignores
  - Pixel pipeline
  - Texture memory
  - Display lists
  - ...
- Display is not part of OpenGL



```

graph TD
    Application --> VA[Vertex assembly]
    VA --> VO[Vertex operations]
    VO --> PA[Primitive assembly]
    PA --> PO[Primitive operations]
    PO --> R[Rasterization]
    R --> FO[Fragment operations]
    FO <--> FB((Framebuffer))
    FB --> Display
    
```

---

---

---

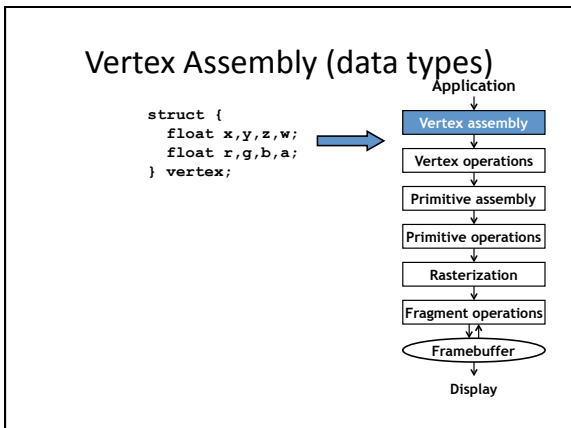
---

---

---

---

---




---

---

---

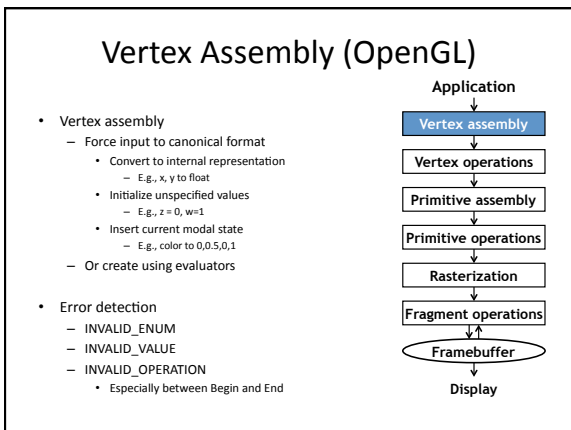
---

---

---

---

---




---

---

---

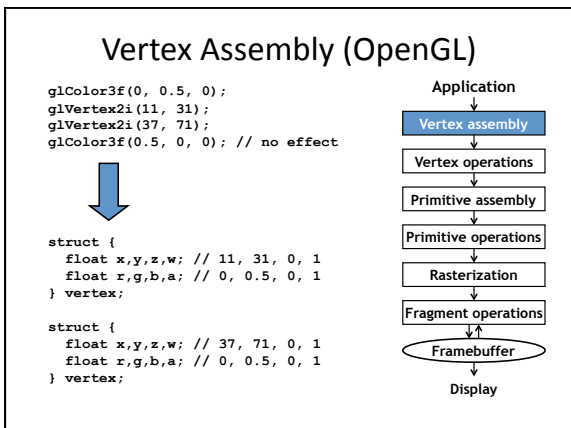
---

---

---

---

---




---

---

---

---

---

---

---

---

### Vertex Operations

- OpenGL
  - Transform coordinates
    - 4x4 matrix arithmetic
  - Compute (vertex) lighting
  - Compute texture coordinates
  - ...
- In our case:
  - Scale (arbitrary 100x100) coordinates to fit window
  - No lighting, no texture coordinates

```

graph TD
    Application --> VertexAssembly[Vertex assembly]
    VertexAssembly --> VertexOperations[Vertex operations]
    VertexOperations --> PrimitiveAssembly[Primitive assembly]
    PrimitiveAssembly --> PrimitiveOperations[Primitive operations]
    PrimitiveOperations --> Rasterization[Rasterization]
    Rasterization --> FragmentOperations[Fragment operations]
    FragmentOperations --> Framebuffer((Framebuffer))
    Framebuffer --> Display[Display]
    
```

---

---

---

---

---

---

---

---

### Primitive Assembly (data types)

```

struct {
    float x,y,z,w;
    float r,g,b,a;
} vertex;

struct {
    vertex v0,v1,v2;
} triangle;
or
struct {
    vertex v0,v1;
} line;
or
struct {
    vertex v0;
} point;
    
```

```

graph TD
    Application --> VertexAssembly[Vertex assembly]
    VertexAssembly --> VertexOperations[Vertex operations]
    VertexOperations --> PrimitiveAssembly[Primitive assembly]
    PrimitiveAssembly --> PrimitiveOperations[Primitive operations]
    PrimitiveOperations --> Rasterization[Rasterization]
    Rasterization --> FragmentOperations[Fragment operations]
    FragmentOperations --> Framebuffer((Framebuffer))
    Framebuffer --> Display[Display]
    
```

---

---

---

---

---

---

---

---

### Primitive Assembly

- OpenGL
  - Group vertices into primitives:
    - points,
    - lines, or
    - triangles
  - Decompose polygons to triangles
  - Duplicate vertices in strips or fans
- In our case:
  - Create two triangles from a strip:

```

glBegin(GL_TRIANGLE_STRIP);
glColor(green);
glVertex2f(0,0); // 0
glVertex2f(1,1); // 1
glColor(red);
glVertex2f(2,2); // 2
glVertex2f(1,1); // 3
glEnd();
    
```

```

graph TD
    Application --> VertexAssembly[Vertex assembly]
    VertexAssembly --> VertexOperations[Vertex operations]
    VertexOperations --> PrimitiveAssembly[Primitive assembly]
    PrimitiveAssembly --> PrimitiveOperations[Primitive operations]
    PrimitiveOperations --> Rasterization[Rasterization]
    Rasterization --> FragmentOperations[Fragment operations]
    FragmentOperations --> Framebuffer((Framebuffer))
    Framebuffer --> Display[Display]
    
```

---

---

---

---

---

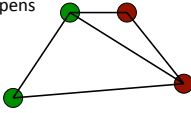
---

---

---

### Primitive Operations

- OpenGL
  - Clip to the window boundaries
    - Actually to the frustum surfaces
  - Perform back-face / front-face ops
    - Culling
    - Color assignment for 2-side lighting
- In our case
  - Nothing happens



Application

↓

Vertex assembly

↓

Vertex operations

↓

Primitive assembly

↓

**Primitive operations**

↓

Rasterization

↓

Fragment operations

↕

Framebuffer

↕

Display

---

---

---

---

---

---

---

---

### Rasterization (data types)

```

struct {
    float x,y,z,w;
    float r,g,b,a;
} vertex;

struct {
    vertex v0,v1,v2
} triangle;

struct {
    short int x,y;
    float depth;
    float r,g,b,a;
} fragment;
    
```

Application

↓

Vertex assembly

↓

Vertex operations

↓

Primitive assembly

↓

Primitive operations

↓

**Rasterization**

↓

Fragment operations

↕

Framebuffer

↕

Display

---

---

---

---

---

---

---

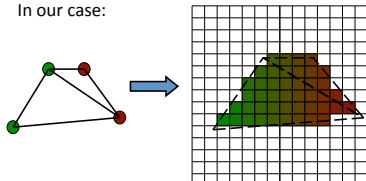
---

### Rasterization

OpenGL

- Determine which pixels are included in the primitive
  - Generate a fragment for each such pixel
- Assign attributes (e.g., color) to each fragment

In our case:



Application

↓

Vertex assembly

↓

Vertex operations

↓

Primitive assembly

↓

Primitive operations

↓

**Rasterization**

↓

Fragment operations

↕

Framebuffer

↕

Display

---

---

---

---

---

---

---

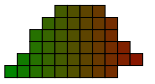
---

### Fragment Operations

OpenGL

- Texture mapping
- Fragment lighting (OpenGL 2.0)
- Fog
- Scissor test
- Alpha test

In our case, nothing happens:



Application

↓

Vertex assembly

↓

Vertex operations

↓

Primitive assembly

↓

Primitive operations

↓

Rasterization

↓

Fragment operations

↕

Framebuffer

↕

Display

---

---

---

---

---

---

---

---

### Framebuffer (2-D array of pixels)

```

struct {
    float x,y,z,w;
    float r,g,b,a;
} vertex;

struct {
    vertex v0,v1,v2
} triangle;

struct {
    short int x,y;
    float depth;
    float r,g,b,a;
} fragment;

struct {
    int depth;
    byte r,g,b,a;
} pixel;
    
```

Application

↓

Vertex assembly

↓

Vertex operations

↓

Primitive assembly

↓

Primitive operations

↓

Rasterization

↓

Fragment operations

↕

Framebuffer

↕

Display

---

---

---

---

---

---

---

---

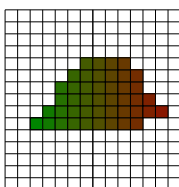
### Fragment ↔ Framebuffer Ops

OpenGL

- Color blending
- Depth testing (aka z-buffering)
- Conversion to pixels

In our case, conversion to pixels:

**Key idea: images are built in the framebuffer, not just placed there!**



Application

↓

Vertex assembly

↓

Vertex operations

↓

Primitive assembly

↓

Primitive operations

↓

Rasterization

↓

Fragment operations

↕

Framebuffer

↕

Display

---

---

---

---

---

---

---

---

Some more detail...

---

---

---

---

---

---

---

---

**Modeling and Viewing**

- OpenGL provides no functions itself for directly specifying a view
  - it has no ‘policy’ for how a ‘camera’ is to be specified
- It provides no data structures for model hierarchies.
- Instead it provides fundamental tools that allow the construction of many different camera models and hierachies.

---

---

---

---

---

---

---

---

**Modelview Matrix**

- A stack of matrices is maintained called the ‘modelview’ stack.
- The current modelview matrix is used to multiply vertices at the first stage of the rendering pipeline
  - equivalent to matrix C.M (OpenGL notation)
    - C = CTM, M: WC->VC
- `glMatrixMode(GL_MODELVIEW)`
  - making changes to modelview

---

---

---

---

---

---

---

---

### Matrix Operations

- `glLoadMatrix{f}{d}(const GLfloat *m);`  
– replaces current matrix
- `glMultMatrix{f}{d}(const GLfloat *m);`  
– if `c` is current matrix then `cm` is the new one
- `glPushMatrix{f}{d}();`  
– pushes copy of current matrix down on stack;
- `glPopMatrix();`  
– restores top of stack to be current matrix.

---

---

---

---

---

---

---

---

### Example: Object Hierarchy

- Suppose the current modelview matrix is `M`:  
`WC->VC` (i.e., based on `VRP, VPN, VUV`).
- `GObject *object; //pointer to graphics object`  
– `glMatrixModel(GL_MODELVIEW);`  
– `glPushMatrix(); // push and duplicate current matrix`  
– `glMultMatrix(object->CTM); // multiply M by CTM`  
– `myDraw( object ); // now draw all faces in object`  
– `glPopMatrix(); //restore original M`

---

---

---

---

---

---

---

---

### The Projection Matrix

- `glMatrixMode(GL_PROJECTION);`  
– subsequent matrix ops affect this stack (only 2 deep)
- A perspective projection can be specified by:-  
– `glLoadIdentity();`  
– `glFrustum(left, right, bottom, top, near, far);`  
• each argument is `GLdouble`

---

---

---

---

---

---

---

---

## Transformations

- `glTranslate{d}{f}(x,y,z);`  
– translation matrix  $T(x,y,z)$
- `glScale{d}{f}(x,y,z);`  
– scaling matrix  $S(x,y,z)$
- `glRotate{d}{f}(angle, x, y, z);`  
– matrix for positive (anti-clockwise) rotation of angle degrees about vector  $(x,y,z)$
- If  $C$  is current matrix, and  $Q$  is transformation matrix, then new current matrix is  $CQ$  (OpenGL notation)

---

---

---

---

---

---

---

---

## Cautions

- OpenGL uses a RH coordinate system throughout (hence the default VPN is the negative z-axis).
- It adopts the convention of points as column vectors and post-multiplication:
- The transpose of all our matrices should be used!
 
$$\begin{pmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

---

---

---

---

---

---

---

---

## Shading and Colours

- Shading properties  
– `glShadeModel(GL_SMOOTH | GL_FLAT)`
- Colour  
– `glColorNT{V}(r,g,b,{a})`
  - $N=3,4$
  - $T=b,s,i,ub,ui,us$
  - $v$  implies passing a pointer to array of colours

---

---

---

---

---

---

---

---

### Lighting and Materials

- Many lighting parameters
- Specify a material for primitives
  - emissive, ambient, shininess, specular
  - GLfloat mat\_spec = { 0.5, 0.5, 1.0, 1.0};
  - glMaterialfv(GL\_FRONT, GL\_SPECULAR, mat\_spec)
  - glColorMaterial(GL\_FRONT, GL\_DIFFUSE)

---

---

---

---

---

---

---

---

### Lights

- Must enable a light with materials
  - GLfloat light\_pos = { 1.0, 2.0, 1.0, 0.0}
  - glLightfv(GL\_LIGHT0, GL\_POSITION, light\_pos)
  - glEnable(GL\_LIGHTING)
  - glEnable(GL\_LIGHT0)

---

---

---

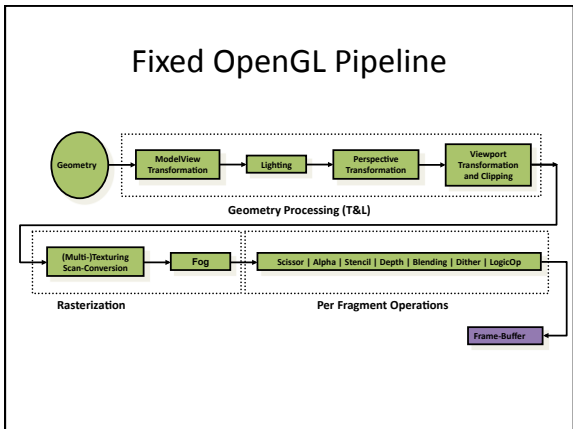
---

---

---

---

---




---

---

---

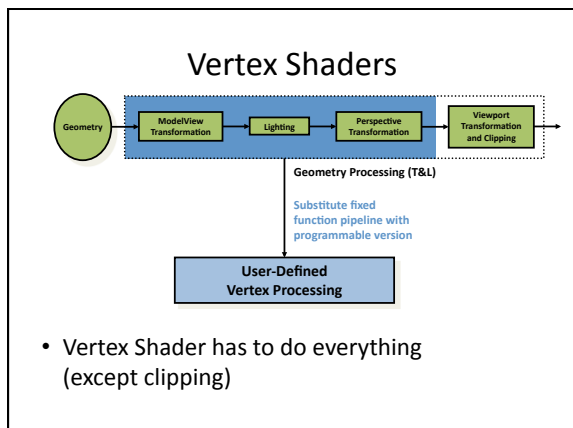
---

---

---

---

---



---

---

---

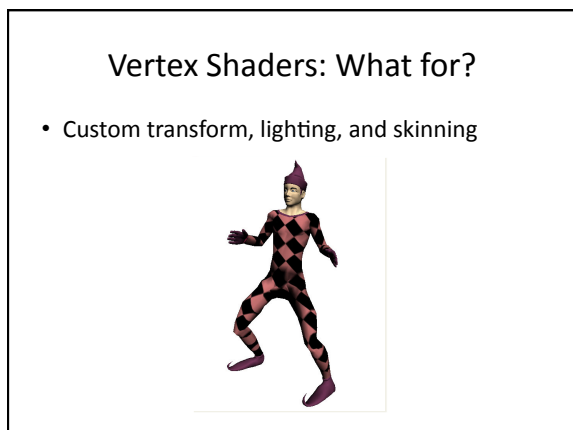
---

---

---

---

---



---

---

---

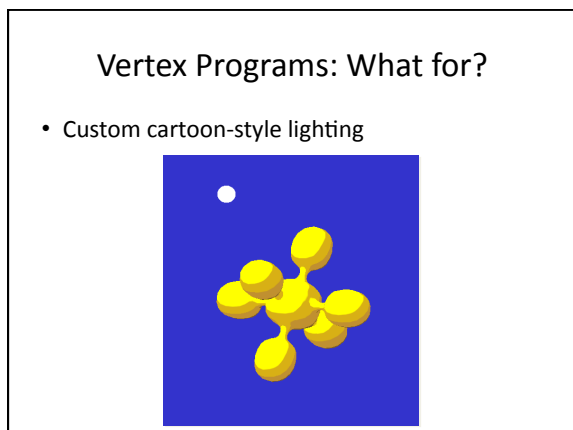
---

---

---

---

---



---

---

---

---

---

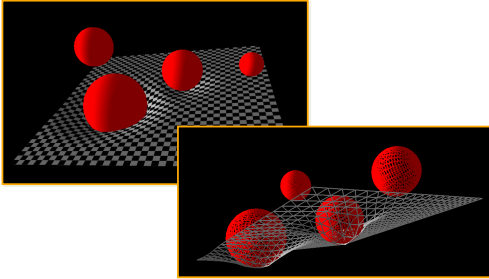
---

---

---

### Vertex Shaders: What for?

- Dynamic displacements of surfaces by objects



---

---

---

---

---

---

---

---

### Vertex Shaders

- What is a vertex shader?
  - A small program running on GPU for each vertex
  - GPU instruction set to perform all vertex math
  - Reads an untransformed, unlit vertex
  - Creates a transformed vertex
  - Optionally ...
    - Lights a vertex
    - Creates texture coordinates
    - Creates fog coordinates
    - Creates point sizes
    - Latest GPUs: can read from texture

---

---

---

---

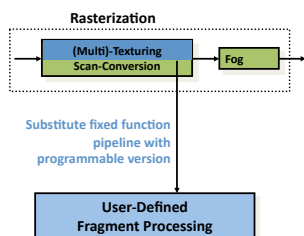
---

---

---

---

### Pixel Shaders



---

---

---

---

---

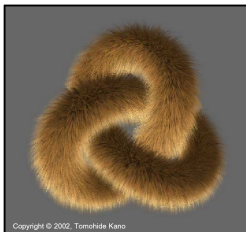
---

---

---

### Pixel Shaders

- Little assembly program for *every* pixel
- What for? E.g., per-pixel lighting



---

---

---

---

---

---

---

---

### Pixel Shaders

- What is a pixel shader?
  - A small program running on GPU for each fragment
  - GPU instruction set to perform math (lin. algebra, ...)
  - Takes a fragment (depth/color) and texture coordinates
  - Creates an output color (or discard)
  - Optionally ...
    - Read texture
    - Perform computation

---

---

---

---

---

---

---

---

### Summary

- OpenGL is an architecture
  - Mechanism (the OpenGL pipeline)
  - Interface (API) to feed and manage the mechanism
- The pipeline is best understood in terms of data types:
  - Vertex
  - Primitive (point, line, triangle)
  - Fragment
  - Pixel

---

---

---

---

---

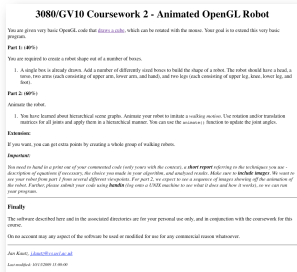
---

---

---

## Coursework

- Second coursework: OpenGL
- Due: **Wednesday 16/Dec/2009**
- Hand in both:
  - Write-up
  - Code ('handin')




---

---

---

---

---

---

---

---

---

---

End

---

---

---

---

---

---

---

---

---

---