CP2 Revision

theme: OpenGL

An overview of OpenGL

- OpenGL transformations
  - viewport transformation
  - projection transformation
  - model & view transformations
  - matrix stacks
- OpenGL primitives & object construction
  - → more advanced topic: vertex arrays & display lists
- OpenGL lighting & materials
  - → CG lighting & reflection models
- OpenGL state management
  - states & attribute groups
  - attribute stacks
  - display lists (*see object construction*)
- OpenGL 2D operations
  - bitmaps
  - pixmaps
  - color-buffer (*of framebuffer*) operations (*including blending*)
- OpenGL textures & texture mapping

OpenGL libraries

- gl - **`gl.h`** (*standard OpenGL library*)
- glu - **`glu.h`** (*OpenGL utility library*)
- windowing system extensions (*glx, wgl, agl ...*)
- glut - **`glut.h`** (*GL utility toolkit*)

OpenGL API

OpenGL naming conventions

- commands (*functions & procedures*) are prefixed by **`gl`**
- datatypes are prefixed by **`GL`**
- constants are prefixed by **`GL_`**

OpenGL command syntax

command prefix + command id + (up to 4) args id characters

- 1**st** character contains number of expected GL parameters
- next character (*pair*) contains expected GL parameter datatype (*see below*)
- last character (***v***) indicates if the parameters will be in an array (*vector*) or on their own

example:

```
void glVertex3f (GLfloat, GLfloat, GLfloat);
void glVertex4iv (GLint*);
```

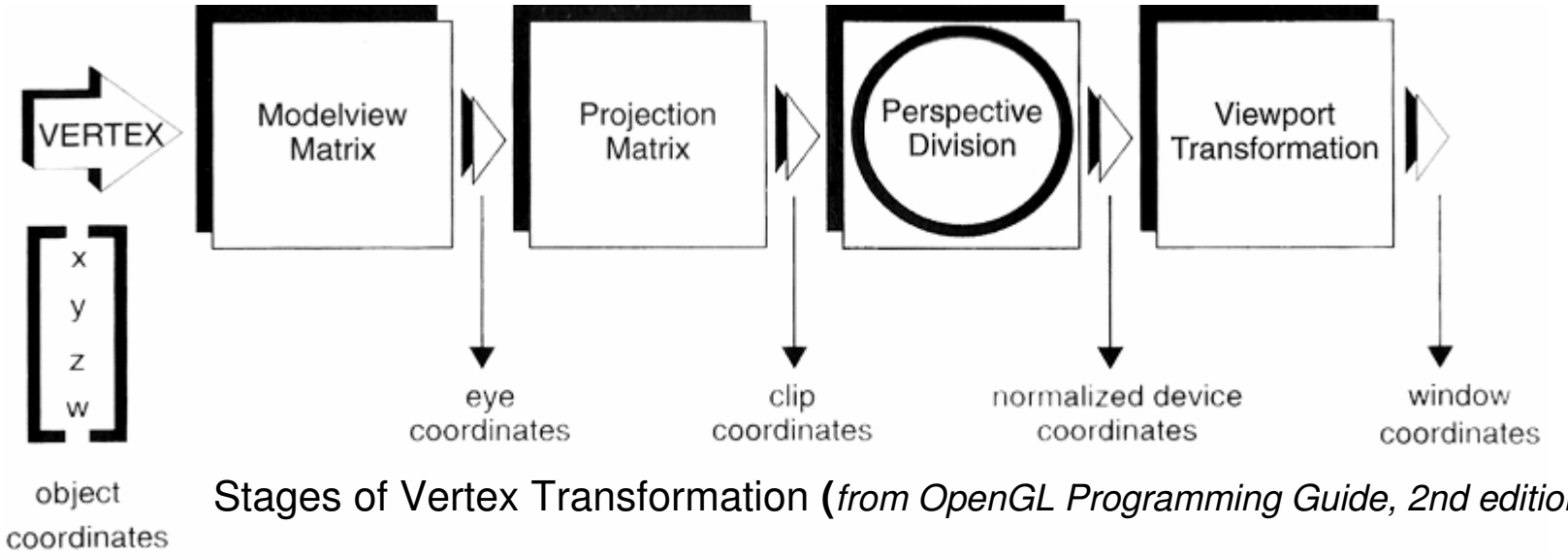
OpenGL datatypes

OpenGL type	ANSI C type	type ID	size
`GLbyte`	`char`	b	8 bit
`GLdouble` `GLclampd`	`double`	d	64 bit
`GLfloat` `GLclampf`	`float`	f	32 bit
`GLint` `GLsizei`	`int / long`	i	32 bit
`GLshort`	`short`	s	16 bit
`GLubyte` `GLboolean`	`unsigned char`	ub	8 bit

`GLuint`
`GLenum`
`GLbitfield`
	`unsigned int /`
	`unsigned long`
		ui	32 bit

`GLushort`	`unsigned short`	us	16 bit

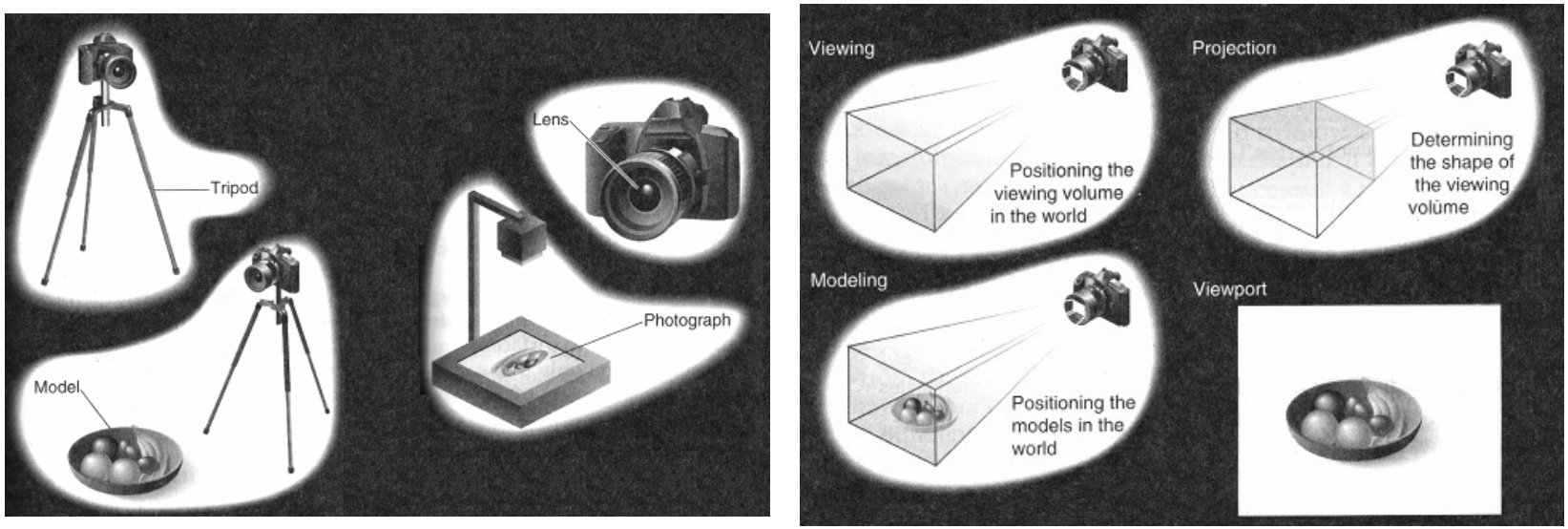
OpenGL rendering pipeline



Stages of Vertex Transformation (*from OpenGL Programming Guide, 2nd edition*)

- viewing transformation (*camera positioning*)
- model transformation (*object positioning*)
- projection transformation (*definition of viewing volume*)
- viewport transformation (*world-space to screenspace*)

OpenGL camera analogy



(*from OpenGL Programming Guide, 3rd edition*)

default camera position & alignment

- position = origin (0,0,0)
- up-vector = y axis (0,1,0)
- direction = - z axis(0,0,-1)

OpenGL transformations

- always affect the currently selected transformation matrix
- immediate mode allows one of three possible transformations:
  - translation
  - rotation
  - scaling

viewing volume is aimed by transformation of the ModelView matrix

- transformations can be called explicitly
- alternatively the utility function **`gluLookAt`** can be used

current transformation matrices are on the top of matrix stacks

- this is mainly useful for managing object hierarchies with objects that keep a relative position between each other

OpenGL transformations (2)

OpenGL Matrix Stacks

- hold transformation matrices
- useful for hierarchical local transformations

Modelview matrix stack allows up to 32 matrices to be stacked
Projection matrix stack allows up to 2 matrices to be stacked
Texture matrix stack allows up to 2 matrices to be stacked

`glPushMatrix();`

- save current matrix on matrix stack
- calling glPushMatrix will store the current transformation matrix

`glPopMatrix();`

- load topmost matrix off matrix stack
- calling glPopMatrix will restore a previously saved matrix

OpenGL transformations (3)

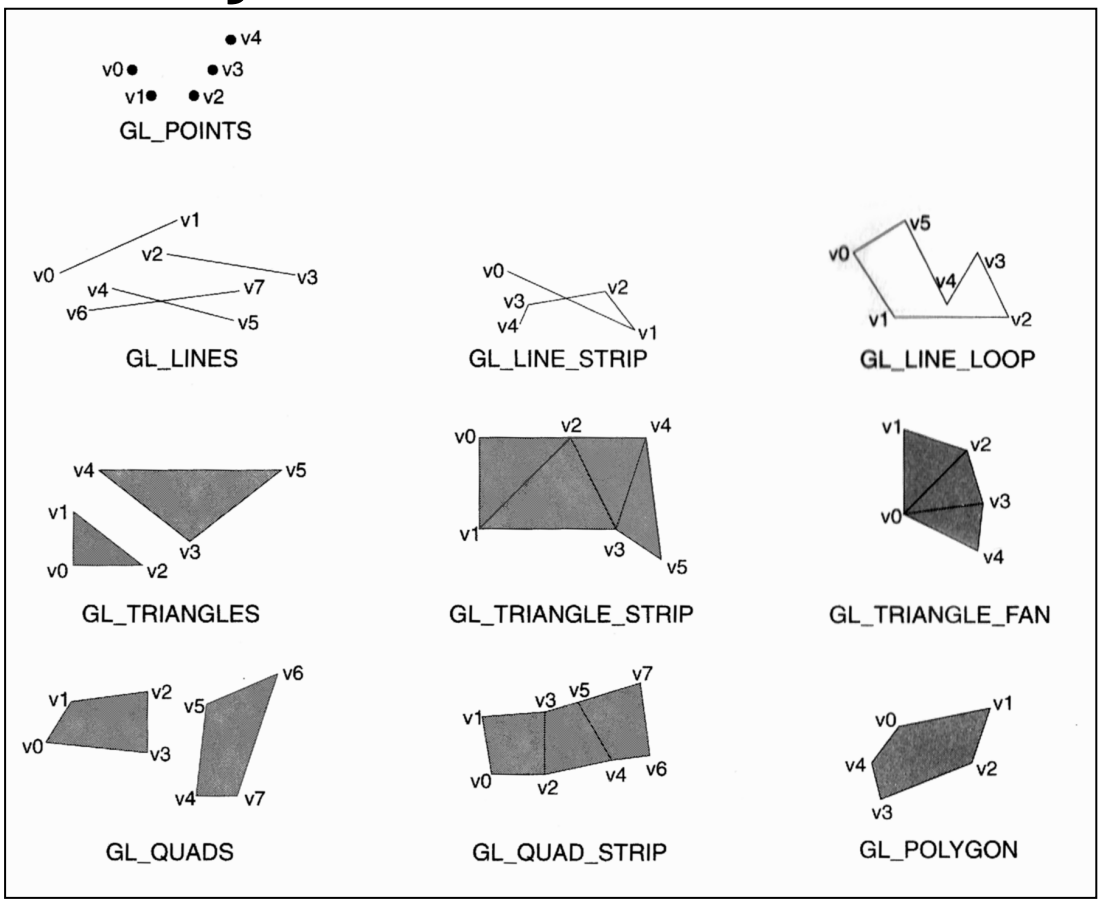
Matrix Stack usage

- transformations are additive (*each new transformation is multiplied with the current transformation matrix*)
- if a number of objects is supposed to use the same base transformation then
  1. the base transformation must be recreated before each object is transformed and drawn – or
  2. the base transformation must be saved before each object is transformed and drawn and then reloaded afterwards – or
  3. The base transformation must be pushed onto the matrix stack before each object is transformed and drawn and then popped off the matrix stack afterwards

this is mainly useful for

- managing object hierarchies with objects that keep a relative position between each other

OpenGL object construction



from OpenGL Programming Guide, 2nd edition

primitives

geometric OpenGL primitives: point, line segment, polygon

defined between a **`glBegin(`*`primitive`*`)`** – **`glEnd()`** block.

OpenGL object management

we know that

- primitives are directly drawn into the frame buffer
- this is done by calling **`glBegin()`** with the primitive type as parameter
- this would be followed by a list of OpenGL commands that specify the vertices that define the object's shape (*+ [optional] normals, colours, texture coordinates*) until a **`glEnd()`** is encountered.

this is:

- simple (*but*)
- not very efficient (*too many function calls*)

better:

- group vertices together so one function can draw whole primitive (*group*)

solution:

- vertex arrays

vertex arrays

- arrays containing vertex data
- can be used with different arrays for vertex, colour, normals, texture coordinates
- or with a single (*interleaved*) arrays that contain all vertex data (*by using* ***glInterleavedArrays***)

OpenGL client must be enabled for Array use:

```
void glEnableCLientState(GLenum arrayTypeID);
```

example (*for vertex and RGBA colour*):

```
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);
```

usage of arrays needs to be declared:

```
glVertexPointer(GLint size,GLenum type,GLsizei stride,GLvoid *pointer);
```

- size - distance (*in values*) between vertices
- type - datatype of vertex data
- stride - byte offset between neighbouring vertices
- pointer - pointer to array

vertex arrays (2)

also available as

- glColorPointer - for RGBA colours
- glIndexPointer - for indext colour mode
- glNormalPointer - normals
- glTexCoordPointer - texture coordinates
- glEdgeFlagPointer - for polygon edges

simplest way to draw by calling

`void glDrawArrays(GLenum primitive_Type,GLint first,GLsizei number);`

draws *number* elements of *primitive_Type* from the defined arrays, starting with *first* element

other methods that access single array elements or element groups (*between* ***glBegin*** *&* ***glEnd***) are also available

OpenGL display lists

the default mode in which OpenGL commands are executed is the "immediate mode" (*IM*):

- commands are sent from the client to the server as soon as they are issued and then executed immediately
- flexible, but inefficient if commands have to be reissued a number of times with identical parameters

this can be optimised by using display lists:

- they are a cache of OpenGL commands that are precalculated which could be used as a “retained mode” (*RM*) for rendering
- they are stored on the OpenGL server (*therefore they do not have to be resent by the client every time*)
- they can greately improve performance if they store often used command sequences

OpenGL display lists (2)

advantages (*pros*):

- can speed up most OpenGL commands that affect the server (*transformations, state changes - including lights and materials, bitmaps etc.*)
- can be nested in hierarchical display lists (*does not nest child lists, but only calls to child lists*)
- up to 2^{32} display lists can be defined (*video RAM size is the limit*)

disadvantages (*cons*):

- short display lists might be less efficient because of overhead that is generated when lists are called
- once created the contents of a display list can no longer be changed
- no feedback possible from within the display list - once it is called it's "***fire and forget***"

OpenGL display lists (3)

steps that are required for using a display list

1. find an unused display list:
   `GLuint glGenLists(GLsizei range);`
   (***range*** *is the numbers of adjacent display lists - function returns first index*)
2. create the display list:
   `void glNewList(GLuint list,GLenum mode);`
   (***list*** *is index of list,* ***mode*** *is* ***`GL_COMPILE`*** *or* ***`GL_COMPILE_AND_EXECUTE`***)
3. call OpenGL commands that should be stored in the list
4. compile the display list:
   `void glEndList(void);`

- executing a display list - **`void glCallList(GLuint list);`**
- check list existence - **`GLboolean glIsList(GLuint list);`**
- deleting lists - **`void glDeleteLists(GLuint list,GLsizei range);`**

OpenGL lighting & materials

CG reflection models (*in general*)

- a light in the commonly used computer graphics reflection model has 3 components that each have three colour values (*red, green, blue*).
- all lighting calculations are per (*raster*) pixel and for each colour (*per-pixel-per-colour*).
- the three light components are ambient light, diffuse light and specular highlights.

The OpenGL lighting model

- light & material have 4 main (*additive*) components:
  - ambient (*background lighting with no direction*)
  - diffuse (*directional light*)
  - specular (*directional highlights*)
  - emissive (*material property **only** - light originating from an object*)

Lighting (*the colour of a vertex*) is calculated by combining the intensity of lights (*light sources*) with the reflection values (*coefficients*) of surface materials (*ambient is one global value – other light components are additive per light-source*)

OpenGL state management

OpenGL states

- values (*attributes*) that are set globally throughout the application
- can be stacked
- can be a single value (*flag*) or a group (*array*) of values (*drawing colour etc.*)
- can be queried

single value states:

switching on:

```
void glEnable(GLenum stateID);
```

switching off:

```
void glDisable(GLenum stateID);
```

complex (*compound*) states:

among others:

- current normal
- currently set colour
- culling mode
- matrix mode
- current viewport
- etc. (*over 200 different states*)

OpenGL state management (2)

OpenGL Attribute Groups and OpenGL Attribute Stack

certain attributes are grouped and can be manipulated or accessed as a group.

the most commonly used are:

- GL_COLOR_BUFFER_BIT (*screen*)
- GL_DEPTH_BUFFER_BIT (*Z buffer*)
- GL_ENABLE_BIT (*all flags that can be enabled or disabled*)

OpenGL has 2 attribute stacks which can be used to save a current snapshot of the states of the OpenGL server and client:

- **`void glPushAttrib(GLbitfield groupID);`**
  `void glPushClientAttrib(GLbitfield groupID);`
  example: **`glPushAttrib(GL_COLOR_BUFFER_BIT);`**

retrieval with

- **`void glPopAttrib(void);`**
  `void glPopClientAttrib(void);`

OpenGL bitmaps & pixmaps

Bitmaps:

- (*rectangular*) 2D array of 0s and 1s
- uses currently selected drawing colour for pixels marked as **`1`**
- commonly used for drawing characters
- select drawing position with **`glRasterPos*`**
- draw with **`glBitmap`**

Pixmaps:

- (*rectangular*) 2D array of multiple-bit (*colour*) data
- several pieces of data per pixel in a range of formats (*typically RGBA, but different pixel alignments available*)
- usually drawn onto colour buffer (*of framebuffer*), but can also be placed into stencil buffer or depth buffer
- can be used as 2D texture maps

OpenGL blending

alpha blending

- done using the Alpha values of RGBA colours for creating transparency
- combining the colour of what is being drawn (*source*) with that of the respective pixels that already exist in the framebuffer (*destination*)
- blending uses the alpha values of source and destination colour as well as selected blending factors for source and destination to calculate the resulting colour in the framebuffer using the blending equation

blending equation:

$$RGBA = (R_sS_r+R_dD_r, G_sS_g+G_dD_g, B_sS_b+B_dD_b, A_sS_a+A_dD_a)$$

OpenGL textures

- (*rectangular*) arrays of data (*like pixmaps*)
- individual pieces of data are called ***texels***

to create a texture these (*5*) steps must be executed:

1. creating a texture object
2. specifying a texture for that object
3. indicating the texture application method
4. enabling texture mapping
5. drawing the scene (*all geometric coordinates will require additional texture coordinates*)

texture mapping requires the framebuffer to use RGBA

OpenGL textures - multitexturing

modern graphics hardware allows multiple textures to be applied to a single face. The number of texture units in the hardware define how many textures can be combined for this – each texture unit can be activated & addressed separately.

to use multi-texturing the following steps must be executed:

1. creating a texture object
2. specifying a texture for all texture units that are going to be used
3. indicating the texture application method for all texture units
4. enabling texture mapping
5. drawing the scene (*all geometric coordinates will require additional texture coordinates for each texture unit*)

OpenGL fog

adds realism to a scene by making objects "*fade*" into the distance

activated by calling **`glEnable(GL_FOG);`**

objects are blended with a fog colour (*should be the background/clear colour*) using a fog blending factor $\mathbf{C} = f\,\mathbf{C_i} + (1 - f)\,\mathbf{C_f}$:

- there are 3 blending factors using a different fog density equation
  1. linear - $f = {}^{(end-z)}/_{(end-start)}$
  2. exponential - $f = e^{-(density*z)}$
  3. exponential^2 - $f = e^{-(density*z)^2}$

C = resulting colour
f = fog blending factor
Ci = original colour
Cf = fog colour