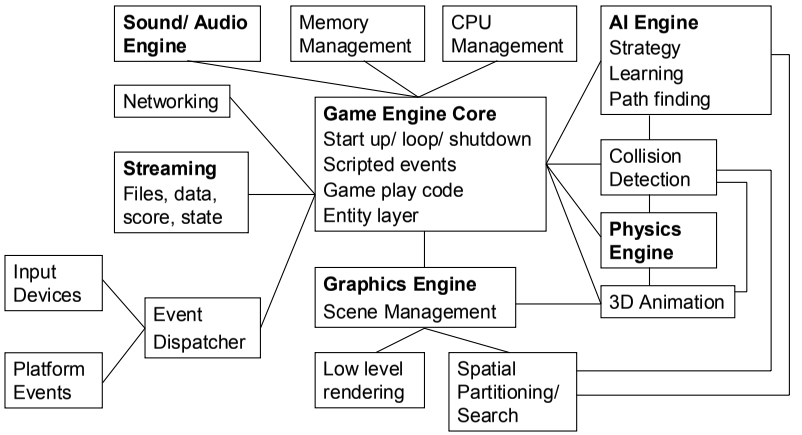
4. Game Engine Design

4.1 The Game Engine in More Detail



The Game Engine in More Detail (cont'd)

Initialization
- Allocate memory
- Load files and data
- Build tables

Game Loop
- Get player input
- Game logic: AI, collision detection, physics
- Render next frame to buffer
- Draw buffer content on screen
- Play sounds (in parallel, sync. thread)
- Check memory and CPU to sync. game frame rate

The Game Engine in More Detail (cont'd)

Menu Loop
- Get player input
- Logic
- Render 2D
- Change settings

Shutdown
- Cleanup
- Save state and score
- Free memory
- Close files
- Exit to OS

4.2 Game Engine Architecture

Reference: Chapters 21 and 22 of "3D Games" textbook

- Game Engine
  - SW tools, or library, that implement basic operations, visibility, collision, rendering, for generic 3D objects.
  - Functionality:
    - Data management, visibility processing, and rendering for generic objects
    - Geometry: polygon, B-Spline meshes, dynamic LOD's
    - Lights and shadows
    - Textures and light maps
    - View control (camera)
    - Input device handling
    - Console I/O
    - Others (sound, multi-player support, special effects)

Game Engine Architecture (cont'd)

- Object-oriented design principles
  - Moderately extensive use of inheritance hierarchy, including multiple inheritance
  - Virtual functions: Declared in a game engine class and implemented by a derived class (class of a plug-in)
  - Simulation state update and rendering traversal defined in game engine for generic objects independently of the actual derived class's implementation
- Generic objects have virtual functions (methods) for performing their actions: initialization, update, clone, draw

Game Engine Architecture (cont'd)

- Plug-in
  - Collection of classes that define objects and their functions derived from a game engine's generic classes.
  - Objects of the plug-in classes have
    - data that extends that inherited from the parent class(es)
    - defined behaviours that override the virtual functions of the parent classes as well as new functions that extend their behaviour.
  - Typical plug-in functions that override parent class virtual functions are: step, ray-intersect, and draw functions.
  - Plug-in's typically developed and saved as DLL files that are loaded at run time by a game front end.

Game Engine Architecture (cont'd)

- Front-end
  - An application program that loads a plug-in (or plug-in's) and invokes game engine functions to run a game or simulation.
  - Includes an instance of the game engine class.
    - typically stored as a public, global pointer to the game engine object.
    - Note: for efficiency many game engine state variables are declared as public to eliminate need for invoking access functions (though use of inline access functions is recommended)
  - The main simulation loop is contained in the front end and it typically calls the engine's step (update) function and then calls the engine's render function.

Game Engine Architecture (cont'd)

- Game Engine Examples
  - GLUT and OpenGL
  - Fly3D, The Nebula Device, Torque, …
  - Quake, DOOM, Unreal, …
- Plug-in Examples
  - Not typically used with GLUT and OpenGL
  - Various Fly3D plug-in's (see "3D Games" book and CD-ROM)
- Front-end
  - Any GLUT/OpenGL application
  - FLY3D: flyFrontend, flyEditor, flyServer

4.3 Example: The Fly3D Engine

- Fly3D design principles
  - Collection of C++ classes. Examples: base_object, boundbox, bsp_node, bsp_object, console, face, flyEngine, mat4x4, particle, picture, plane, render, sound, vector
  - Front end instantiates a single engine object stored as a global pointer variable, e.g., **flyEngine* engine = new flyEngine();**
  - A front end typically loads an initialization file that defines the plug-in(s) to be used and additional files such as texture maps, sound files, BSP files.

The Fly3D Engine (cont'd)

- Example (somewhat simplified for sake of learning)
  - **flyEngine** class has a data member that is a pointer to an object of type **bsp_object**, **bsp_object *active_obj;**
  - **bsp_object** class is derived from class **particle** and has a data member, **bsp_object *next_elem;** so it may be used as an element in a linked list.
  - **bsp_object** class has virtual functions, **int step(int dt)** and **void draw();**
  - **flyEngine** class has funct., **void flyEngine::step(int dt)**
  - Front end, such as **flyFrontEnd**, loads an init. file that specifies the following to be performed:
    - Load plugin DLL with code for classes derived from **bsp_object**
    - Construct object(s) and assign them to **engine** linked list
  - **flyFrontEnd** contains code that calls **engine→step()** which traverses the active object linked list calling each object's **step()** function and also code that determines visible objects, then calls their **draw()** function.

The Fly3D Engine (cont'd)

- Front end main loop:
  - Compute dt, elapsed time since last frame in msec
  - Call flyengine→step(); // calls step(int dt) function
  - Call render→DrawView();
- The engine's step(int dt) function:
  - Perform input and other updates (console commands, light maps)
  - For each active object
    - If object life is over
      - Destroy object and remove from BSP tree and linked list
    - Else
      - Call object->step(dt)
      - If object changed position, reposition in BSP tree
  - Perform other updates (light maps, send messages to plug-in's, if multi-player mode process multi-player messages)

The Fly3D Engine (cont'd)

```
int flyEngine::step()
{
   static int dt,t0; // at init t0=timeGetTime()-start_time;

   // compute elapsed time
   cur_time=timeGetTime()-start_time;
   dt=cur_time-t0;
   if (dt>0){
       t0=cur_time;
       if (dt<1000){
             step(dt);
             return dt;
       }
   }
   return 0;
}
```


The Fly3D Engine (cont'd)

```
void flyEngine::step(int dt){
  { ... }                      // Do some update stuff
  bsp_object *o=active_obj0;   // pointer to 1st active object
  // loop on all active objects
  while(o){
      if (o->life<0)           // if object life is over,
           { . . . }           // destroy it, remove from BSP
      else {                   // step object
            if (o->step(dt))   // returns 1 if object moved
                  { . . . }    // reposition in BSP-tree.
           }
      o=(bsp_object *)o->next_obj;
  }
  { ... }                      // Do some other update stuff
}
```


The Fly3D Engine (cont'd)

- render->DrawView()
  - Calls InitView() // sets current rendering and lighting parameters
  - Clears window background (glClear with appropriate bits set)
  - Sends **FLYM_DRAWSCENE** message to all plug-in's
  - Plug-in processes draw message:
    - Sets view,
      flyengine->set_camera(flyengine->cam);
    - Draws scene (all bsp_objects accepted by BSP processing),
      flyengine->draw_bsp(bsp);
      bsp (pointer) is BSP tree root node, **bsp_node* bsp;**
    - The BSP (Binary Space Partitioning) tree subdivides scene space such that during tree traversal bsp_objects that are totally outside the view frustum are excluded from drawing and those inside or partially inside are drawn.

The Fly3D Engine (cont'd)

- An overview of the Fly3D game engine classes, plug-in's, and front ends
  - Engine classes (some selected examples)

vector	face	light_map
mat4x4	base_object	picture
boundbox	particle	console
plane	bsp_node	flyEngine
vertex	bsp_object	

  - Plug-in's (selected examples)
    - **gamelib.dll:** sound, particle_system, sphere, explode
    - **lights.dll:** dynamic light, spotlight, spritelight, meshlight
    - **panorama.dll:** implements texture mapped "sky box"

The Fly3D Engine (cont'd)

- Plug-in's (selected examples cont'd)
  - **weapon.dll**
  - **menu.dll**
  - **ship.dll**
  - **walk.dll**
- Front ends (selected examples)
  - **FlyFrontend** – menu selection for walk, car, or sip game
  - **FlyEditor** – object and level editor
  - **FlyServer** – for multi-player game

4.4 Game Engine Programming with OpenGL under Windows

- Rendering Goal:
  - Fast display (~30-60 Hz) AND realistic visual effects
- Basic simulation steps:
  - Initialize: Build or load 3D geometry and set initial state (for desktop simulations must also initialize window)
  - Update loop ("forever"): handle inputs, update state, render. If multi-player, also receive and handle network inputs and send updated state to other players (network update rate < rendering rate)
- Applications
  - Video games and VR: entertainment, training
  - Flight/vehicle simulators: training, R&D
- References
  - Win32 and OpenGL tutorials on course Tutorials web page

OpenGL under Windows (cont'd)

- Initialization step
  - Initialize window and graphics: Win32, OpenGL
  - Load game description file
  - Initialize game state, including loading other files (3D object models, texture maps, light maps)
- Simulation update loop
  - Handle inputs
    - Window system inputs
    - Game interaction inputs
  - Compute elapsed frame time and update game state ("step")
  - Perform visibility computations (culling, LOD) and render

OpenGL under Windows (cont'd)

- Program structure using GLUT (CompSci 372)
  - Initialize API
  - Create window
  - Register callback functions
  - Initialize application
  - GLUTMainLoop
- Example GLUT main program:

OpenGL under Windows (cont'd)

```
#include <glut.h>                    // spec file for GLUT library
                                     // (this file has #include <GL/gl.h>)
int main (int argc, char **argv)
{
  glutInit( &argc, argv );           // initialization
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
  glutInitWindowSize( WIN_WIDTH, WIN_HEIGHT );
  glutInitWindowPosition( WIN_X, WIN_Y );
  glutCreateWindow( argv[0] );       // create window
  glEnable(GL_DEPTH_TEST);

  glutMouseFunc( myMouseCallback );  // register callback function
  glutKeyboardFunc( myKeyboardCallback );
  glutReshapeFunc( myReshape );
  glutDisplayFunc( myDraw );         // the draw callback

  myInit();                          // application initialization

  glutMainLoop( );                   // loop forever (handles callbacks)
  return 0;
}
```


OpenGL under Windows (cont'd)

- Win32 basics
  - Creating a window in a Win32 application
    - Initialize a Window class
    - Register the Window class
    - Create the window and get handle to window (`HWND`)
  - Simulation loop in a Win32 application
    - Get the messages from the `WndProc`.
    - Translate and dispatch (act on) the messages
  - Requires 1 callback function to handle all Windows messages

```
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
```

  - May also handle messages elsewhere, e.g.

```
short GetKeyState(int virtualKey);          // prototype
if ( GetKeyState(VK_UP)&0x80 || GetKeyState('W')&0x80 )
```


OpenGL under Windows (cont'd)

- OpenGL initialization from Win32
  - Much more complicated than GLUT
    - Init. and register Window class, create window possibly setting full screen mode using `DWORD` values (display style)
    - Get/set the window rectangle, type `RECT` drawing area)
    - Get the `HDC` (Handle to Device Context)
    - `SetPixelFormat` using `PIXELFORMATDESCRIPTOR` i.e., bit planes and depth (RGBA, other buffers) (similar to glutInitDisplayMode())
    - Set OpenGL viewport and view projection parameters
  - Reference: CompSci 707 OpenGL Tutorial #2 web pages

OpenGL under Windows (cont'd)

- Simulation time
  - Time used in equations that model the simulation behaviour
  - Example: movement of objects (one simulation "time step") $x(t+\Delta t) = x(t) + \Delta x$
- Real time
  - Actual execution time, a.k.a. "wall clock" time
  - Program execution progresses thru a repeating cycle: handle inputs, perform simulation calculations, compute visibility, render, while executing instructions at CPU speed.
  - Execution time in milliseconds may vary from one cycle to the next:

OpenGL under Windows (cont'd)

- Naïve (stupid) motion control uses simulation results in real time
- Correct solution
  - Measure actual frame time: currentTime = GetTickCount()
  - Use elapsed time variable, dt = currentTime – previousTime, then assign previousTime = currentTime
  - Thus, simulation time = real time (but, with 1 frame delay)

.NET Environment

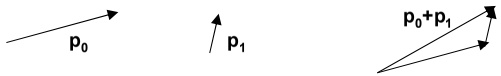
- .NET on university computers with AFS files:
  Cannot write large files in the Debug folder on AFS!
  Thus, you'll have to create project on C:\Usertmp and when done save results into your AFS drive folder.
  Be sure to delete Usertmp files and empty trash!
- In .NET, create a "Win32 Project" (not a console project)
  This will automatically create `_tWinMain`, the `WndProc` callback function, and several other functions. You'll only need to add the OpenGL function calls.
  - You need not create many separate functions as in the GameTutorials.com examples. One long WinMain is acceptable (though poor design from SW Eng. principles).

4.5 Mathematics Review for 3D Geometry

- Coordinate Systems
  - RHS for object model generation, e.g., glVertex*(…)
  - LHS for rendering: OpenGL automatically converts in GL_PROJECTION matrix
- Vertices and faces
  - Vertex = 3D pt (x,y,z),4D Homogeneous Coordinate pt (x,y,z,1)
  - Face = triangle or polygon defined by sequence of 3 or N vertices
  - Triangle guaranteed planar. Polygon may be non-planar.
  - Winding Order
    - Clockwise (CW) or Counter-clockwise (CCW) order determines front or back facing wrt view point.
    - OpenGL assumes CCW ➔ front facing

Mathematics Review (cont'd)

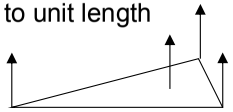
- Vectors:
  - Representation: 3D (x,y,z) or 4D (x,y,z,0)
  - Addition, subtraction: add/subtract each x, y, z component
    $p_0 = (x_0,y_0,z_0)$, $p_1 = (x_1,y_1,z_1)$, $p_0 + p_1 = (x_0+x_1, y_0+y_1, z_0+z_1)$

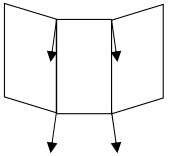


  - Scalar multiplication, division: mult/divide each x, y, z component by the scalar
    $p_0 = (x_0, y_0, z_0)$, a $p_0 = (a\ x_0, a\ y_0, a\ z_0)$
  - Magnitude (length): $| p_0 | = sqrt(x_0 * x_0, y_0 * y_0, z_0 * z_0)$
  - Normalized vector (unit length): $p_0' = (x_0 / | p_0 |, y_0 / | p_0 |, z_0 / | p_0 |)$

Mathematics Review (cont'd)

- Normal Vector
  - vector that is perpendicular (orthogonal) to a face or surface and normalized to unit length





May be computed using cross product

- Planes and plane equation
  - All points (x,y,z) that satisfy the plane equation, $a*x + b*y + c*z + d = 0$ lie in an infinite plane
  - Vector (a,b,c) is a vector with same direction as plane's normal vector.
  - d is distance from origin

Mathematics Review (cont'd)

- Distance between two points
  dist = $\text{sqrt}((x_0-x_1)*(x_0-x_1) + (y_0-y_1)*(y_0-y_1) + (z_0-z_1)*(z_0-z_1))$
- Cross Product
  $\mathbf{v} \times \mathbf{w} = (v_y*w_z - v_z*w_y, v_z*w_x - v_x*w_z, v_x*w_y - v_y*w_x)$
  - If $\mathbf{v}_1$ and $\mathbf{v}_2$ are 2 vectors determined by 3 adjacent vertices of a planar polygon, then $\mathbf{n}=\mathbf{v}_1 \times \mathbf{v}_2$ is the normal vector to that face
- Dot Product (a.k.a. inner product)
  $\mathbf{v} \cdot \mathbf{w} = (v_x*w_x + v_y*w_y + v_z*w_z) = |\mathbf{v}|\,|\mathbf{w}| \cos\theta$
  - where θ is the angle between the 2 vectors
  - If both $\mathbf{v}$ and $\mathbf{w}$ are normalized, $\mathbf{v} \cdot \mathbf{w} = \cos\theta$

Mathematics Review (cont'd)

- Significance of cross product and dot product
  - Cross Product
    - Given 3 points that lie in a plane, vector cross product is a vector whose direction is normal to that plane.
    - Normalize it and you have the normal vector to the plane.
    - The (x,y,z) components of the normal vector are the a, b, c coefficient's of the plane equation for that plane. To solve for the d coefficient, pick any (x,y,z) point in the plane and solve: $d = -(a*x + b*y + c*z)$
  - Dot Product
    - cos **θ** used to determine front/back facing
  - Signed distance of point P=(x,y,z) from plane (a,b,c,d)
    - Substitute (x,y,z) into: dist = $a*x + b*y + c*z + d$ (only if $|(a,b,c)|=1$)
    - dist = 0 ➔ point in plane, dist != 0 is distance from plane either in front (positive) or behind (negative)