How do game engines work?

• The infinite loop:
  ```cpp
  while (!exit) {
  }
  ```

• What things do we do in this loop?
  – Process input
  – Process output
What input and output?
What input and output?

• On each iteration:
What input and output?

- On each iteration:
  - Process mouse, keyboard, joystick, network events
What input and output?

- On each iteration:
  - Process mouse, keyboard, joystick, network events
  - Calculate amount of time that has passed and update the current state of the game
What input and output?

• On each iteration:
  – Process mouse, keyboard, joystick, network events
  – Calculate amount of time that has passed and update the current state of the game
    • older games used to ignore time calculations!
What input and output?

• On each iteration:
  – Process mouse, keyboard, joystick, network events
  – Calculate amount of time that has passed and update the current state of the game
    • older games used to ignore time calculations!
  – Render the frame based on the new game state
Old School Game Design
Old School Game Design

• How did those old text-adventures work anyway?
Old School Game Design

• How did those old text-adventures work anyway?
  – Same concept:
Old School Game Design

• How did those old text-adventures work anyway?
  – Same concept:
    
```c
while (!exit) {
```
Old School Game Design

• How did those old text-adventures work anyway?
  – Same concept:
    while (!exit) {
    read keyboard input
Old School Game Design

- How did those old text-adventures work anyway?
  - Same concept:
    ```
    while (!exit) {
      read keyboard input
      parse input (check to see if player is quitting)
    }
    ```
Old School Game Design

• How did those old text-adventures work anyway?
  – Same concept:
    while (!exit) {
      read keyboard input
      parse input (check to see if player is quitting)
      update state in game
Old School Game Design

• How did those old text-adventures work anyway?
  – Same concept:
    
      while (!exit) {
        read keyboard input
        parse input (check to see if player is quitting)
        update state in game
        print output
      }
Old School Game Design

• How did those old text-adventures work anyway?
  – Same concept:
    while (!exit) {
      read keyboard input
      parse input (check to see if player is quitting)
      update state in game
      print output
    }

Old School World Models

• World was modeled as a tree of objects

• Parsed input triggered function calls on objects:
  with before [;
    Eat: “You feel sick after eating the moldy bread.”;
  ]
Engine Overview

• Object Oriented--everything is in objects and design patterns used heavily

• Plug-in architecture: We tell Ogre what libraries to load at run-time
  – Different scene management objects, platform managers, rendering subsystems, etc
Engine Overview, cont.

- Virtual file system for resources
  - Easily load textures, meshes, etc, from disk to memory

- Scriptable: we can execute scripts in response to events or have scripts execute C++ code
Game Loop Continued

```c
while (!exit) {
    if (!fireFrameStarted())
        break;
    updateAllRenderTarget();
    if (!fireFrameEnded())
        break;
}
```
FrameListener

• How can we hear those frame events?
  – We extend FrameListener
    • Has a couple of important methods
      – virtual bool frameStarted(const FrameEvent &evt)
      – virtual bool frameEnded(const FrameEvent &evt)
    • We define those methods to figure out how to act each iteration through the game loop
  – We then add the listener with the root object
    – Root::addFrameListener(FrameListener *newListener)
Design Patterns

• You’ve probably heard of these before…
• Three very useful ones: Singleton, Façade, State
  – Singleton: allows only a single object
    • #include <OgreSingleton.h>
    • two member functions:
      – static T &getSingleton()
      – static T *getSingletonPtr()
    • Singleton is a template class:
      – class myClass : public Singleton<myClass>
Fascade

• What is a fascade?
  – An interface that hides the underlying complexity of the subsystems
  – Example?
    • A compiler: we call compile methods and it calls the lexer, parser, code generator, etc.
    • In Ogre?
      – Root: this is a top-level singleton object that hides some of the complexity of the programming interface
State

- A method for hiding the complexity of a state machine
  - We have a Context class, that holds a pointer to the current state
  - We have an abstract State class that has virtual functions, such as handle() to process common state requests
  - Context has a private changeState method that State can call to give context a new state
3D Graphics

CIS 399 - Intro to Game Programming
Vectors and Points

• A vector is represented in space by arrows and magnitudes (draw picture)
• A point is represented by the addition of a reference point (perhaps the origin) and a vector
• P, Q, R - all points
• u, v, w - all vectors
Rendering 3D

• In order to understand how we render, we consider the ‘pinhole’ camera
  – pinhole is located at origin, facing z positive
  – film is located distance d behind the pinhole
  – to calculate (xp, yp) on film, we have
    • yp = -y/(z/d), xp = -x/(z/d)
Relationship between Points and Vectors

• We have two equivalent operations that relate points and vectors (demonstrate on board):
  – \( v = P - Q \)
  – \( P = v + Q \)

• Thus, we can show \((P - Q) + (Q - R) = P - R\)
  – \( P - Q = u, Q - R = v, P - R = u + v \)
Lines

• Consider all points of the form:
  – \( P(\alpha) = P_0 + \alpha d \)

• What if we have two points?
  – \( P = Q + \alpha v \), we can find the point \( R \) such that
    • \( P = Q + \alpha(R - Q) = \alpha R + (1 - \alpha)Q \) so that
    • \( P = \alpha_1 R + \alpha_2 R \), where \( \alpha_1 + \alpha_2 = 1 \)
  – (Draw picture to demonstrate points \( Q, R \) and \( P(\alpha) \)
Points and Vectors in Graphics

• We represent vectors by 3 linearly independent vectors (take linear algebra)
  – \([ x \ y \ z ]\) (sideways)

• But then, how do we represent points?
  – \([ x \ y \ z ]\) looks like a vector too!

• We include the point of reference and call this ‘homogenous coordinates’
Frames

• We need vectors to have a point of reference, \( P_0 \)
• We assume that \( 0 \cdot P_0 = 0 \) and \( 1 \cdot P_0 = 1 \)
• And so points are now represented by including their point of reference as such:
  – \([ x \ y \ z \ 1 \ ]\) are points
  – \([ x \ y \ z \ 0 \ ]\) are vectors
Linear Transformations

• A transformation is a function that takes a point or vector and maps it to another point or vector, respectively.
  – A Linear transformation is the restriction that:
    • $f(\alpha p + \beta q) = \alpha f(p) + \beta f(q)$: which means that if we know the transformations of vertices, we can obtain the transformation of linear combination of vertices by linear combinations of the transformation of vertices.
Transformation Matrices

• $v = Au$, where $u$, $v$ are points or vertices, $A$ is the transformation matrix

• (draw a 4x4 matrix)

• Translation (easiest):
  – $[1 \ 0 \ 0 \ x']$
  – $[0 \ 1 \ 0 \ y']$
  – $[0 \ 0 \ 1 \ z']$
  – $[0 \ 0 \ 0 \ 1]$
Transformations Continued

• Inverse of translation is specified by negative x’ y’ z’.

• Scaling: making objects bigger or smaller, or reflecting them
  – \([ x' 0 0 0 ]\)
  – \([ 0 y' 0 0 ]\)
  – \([ 0 0 z' 0 ]\)
  – \([ 0 0 0 1 ]\)
Rotations

- Consider two dimensional case: (draw xy axis, (x, y) with theta, (x’, y’) with sigma between the two lines and points
  - \( x = p \cos \theta \)
  - \( y = p \sin \theta \)
  - \( x’ = p \cos (\theta + \sigma) \)
  - \( y’ = p \sin (\theta + \sigma) \) thus:
    - \( x’ = p \cos \sigma \cos \theta - p \sin \cos \sigma \sin \theta = x \cos \theta - y \sin \theta \)
    - \( y’ = p \cos \sigma \sin \theta + p \sin \sigma \cos \theta = x \sin \theta + y \cos \theta \)
Rotations continued

• Thus we get
  – \([x'] = [\cos \theta \ - \sin \theta] [x]\)
  – \([y'] = [\sin \theta \ \cos \theta] [y]\)

• Note that this is around the z-axis in two dimensions!
  – For x axis:
    – \([1 \ 0 \ 0 \ 0]\)
    – \([0 \ \cos t \ - \sin t \ 0]\)
    – \([0 \ \sin t \ \cos t \ 0]\)
    – \([0 \ 0 \ 0 \ 1]\)
  – For y axis:
    – \([\cos t \ 0 \ \sin t \ 0]\)
    – \([0 \ 1 \ 0 \ 0]\)
    – \([-\sin t \ 0 \ \cos t \ 0]\)
    – \([0 \ 0 \ 0 \ 1]\)
Ogre Scenes

- Handled by the Scene Manager
  - `getRootSceneNode(void or String)`
- Each SceneNode has its own transformation matrix
- The scene is represented hierarchically
- Scene nodes can also add child nodes
  - `createChildSceneNode()`
Scene Nodes

- Scene nodes keep track of the location and orientation of all the attached objects
  - Every scene has a root node, retrievable by `getRootSceneNode()` from your SceneManager.
  - We attach our objects to scene nodes here--so we call `createChildSceneNode("MyNode")` to create a child node off of the root scene node.
What can we do with Scene Nodes?

- `getPosition`, `setPosition` or `translate`
- `scale`, yaw (z), roll(y), pitch(x), `setOrientation` and `getOrientation`
- `attachObject`: here we attach entities
- `getAttachedObject`: by index or its name
- `detachObject`, `detachAllObjects`, `numAttachedObjects`
What do we attach to Scenes?

- Entities! An entity is any movable discrete object.
  - Look at the Ogre hierarchy, and Entity extends Mesh
  - So Ogre must look at the Scene graph, and then draw everything in it by walking through the children, and then drawing all the entities that are visible
Entities

- We can setVisible to make it visible or not, call isVisible to check its status
- getName, returns its name
- getParentSceneNode gets its parent node
- We create them through the SceneManager, which we can call createEntity(“name”, “filename”)
Ogre Cameras and Viewports

• Recall the pinhole camera
  – We can use the Ogre camera just like a pinhole camera

• Ogre Camera API:
  – First, create the camera:
    • mCamera = mSceneMgr->createCamera(“PlayerCam”)
  – Next, set its position and direction
    • mCamera->setPosition(Vector3(0,10,500));
    • mCamera->lookAt(Vector(0, 0, 0));
Cameras Continued...

- We can also set up the camera with other parameters:
  - roll (const Radian &angle) - anticlockwise around its local z-axis
  - yaw (const Radian &angle) - anticlockwise around its local y-axis
  - pitch (const Radian &angle) - anticlockwise around its local x-axis
  - setNearClipDistance (Real nearDist)
  - setFarClipDistance (Real farDist)
  - setAspectRatio(Real ratio) - x/y
  - setFOVy(const Radian &fovy) - y dimension of field-of-view
Viewports

• The viewport is where a scene is rendered to. Typically, we have one viewport and the camera is associated with it (in essence the ‘film’ of the camera)
  – Viewport *vp = mWindow->addViewport(mCamera)

• What can we do with it?
  – vp->setBackgroundColor(ColourValue(0, 0, 0)); //set background to black
  – mCamera->setAspectRatio(Real(vp->getActualWidth())/Real(vp->getActualHeight())))