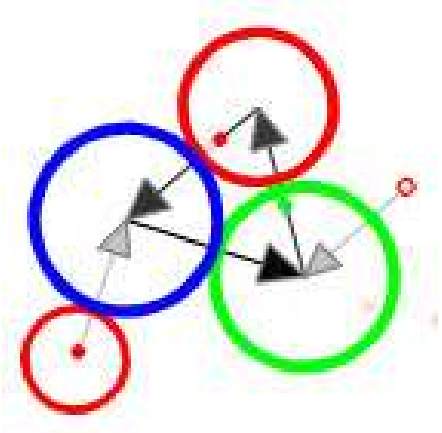


CS 672: Spring 2010

Game Programming and Design

Images:
Chaim Gingold / Chris Hecker
www.slackworks.com/~cog

Game programming patterns

MVC for games

Game Programming Patterns

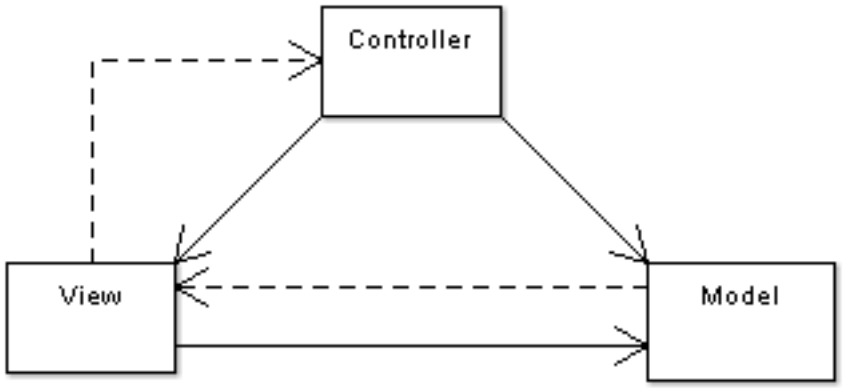
Sources

- **Game Programming Patterns** taken/adapted from
  - Zachary Booth Simpson
    http://www.mine-control.com/zack/patterns/gamepatterns.html
  - Inspired by *Design Patterns* [Gamma et al. 1994]
    http://www.dofactory.com/Patterns/Patterns.aspx

Model View Controller (MVC)

MVC Architecture

- **Model**: The **domain**-specific representation of the information on which the application operates. Data is to be encapsulated by the Model.
- **View**: Renders the model into a form suitable for interaction
- **Controller**: Processes and responds to events, typically user actions, may invoke changes on the model.

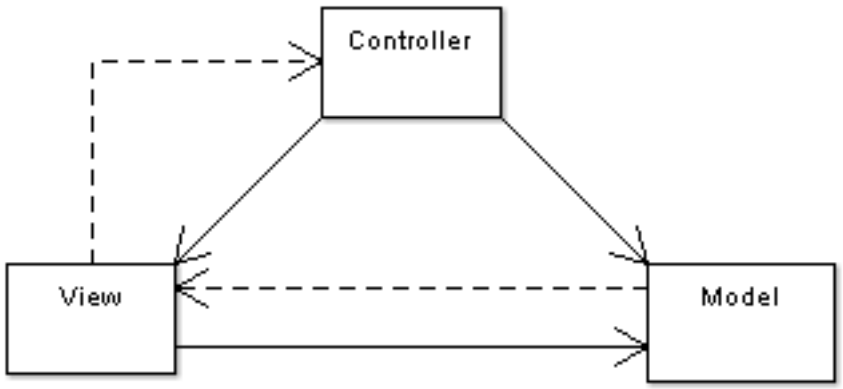


http://en.wikipedia.org/wiki/Model-view-controller

Model View Controller (MVC)

A Typical Case

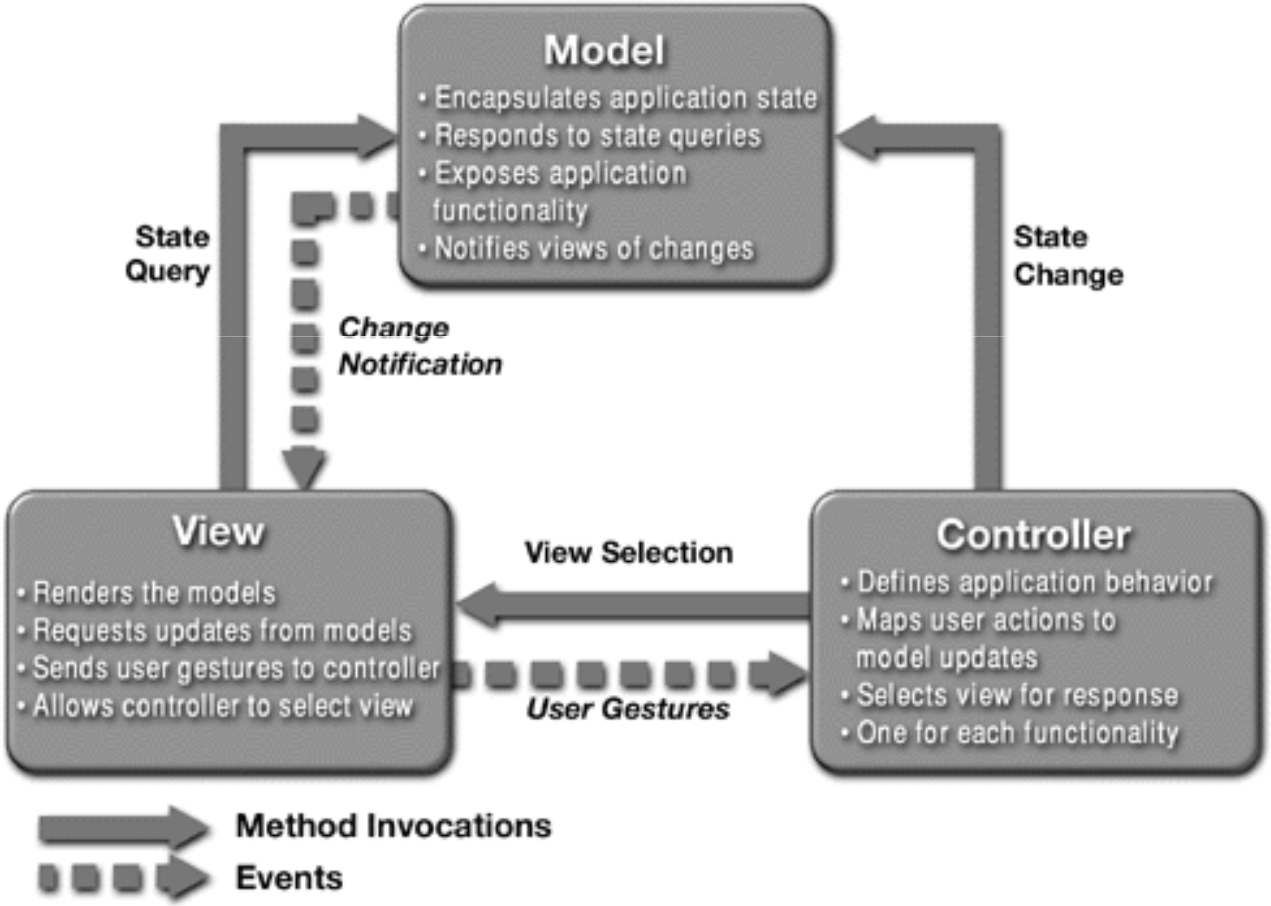
- The user interacts with the user interface in some way (e.g., user presses a button)
- A controller handles the input event from the user interface, often via a registered handler or callback
- The controller accesses the model, possibly updating it in a way appropriate to the user's action
- A view uses the model to generate an appropriate user interface
- Repeat...



http://en.wikipedia.org/wiki/Model-view-controller

Model View Controller (MVC)

Another View



http://java.sun.com/

Main (Game) Loop

- How to Implement MVC in a real-time setting?
- Solution: **Mini Kernel**

```
void updateWorld() {
   for( int i=0; i<numTanks; i++)
   {
      if( tanks[i] )
      {
         updateTankPhys(tanks[i]);
         updateTankAI(tanks[i]);
      }
   }

   for( i=0; i<numSoldiers; i++ )
   {
      ... etc ...
   }
```

```
class BaseController {
  virtual void update() = 0;
}
class MissileController :
    BaseController {
  Model &missle, &target;
  virtual void update(){
   missile.pos += missile.vel;
   missile.vel += (target.pos -
   missile.pos).norm() * missAcc;
  }
}
void miniKernelDoAllControllers(){
    foreach controller in list {
      controller.update();
    }
}
```

Main (Game) Loop

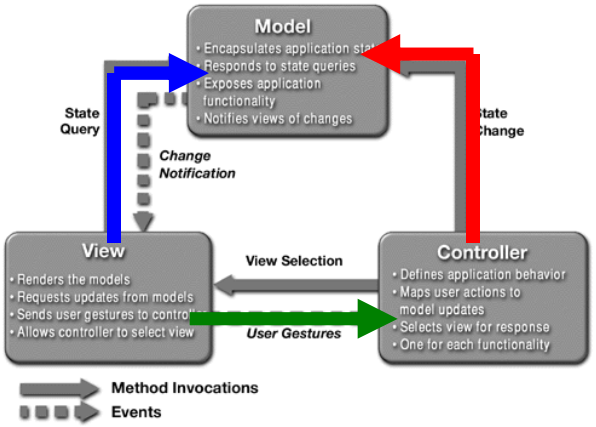
C++ Example

```
bool CGameEngine::RunFrame(GameTime gameTime) {

    GetInput();

    if (gameTime->HasTickPassed()) {    // for each frame
      // move stuff
      miniKernel->RunProcesses(gameTime);
      colManager->resolveCollisions(); // resolve collisions
    }

    // update camera
    camera->Update();
    // render stuff
    renderer->RenderScene();
}
```

Notifies controllers via callbacks

Run each controller in the minikernel

Draw the Model



Model (1)

Model

- **Also Known As.** Database Records, World Items, Item Database
- **Intent.** Store the state of an object which exists in the game world.
- **Motivation.** Each object tracks its state as the game progresses. Game rules define the transition of these states (Controller pattern).
  - Examples of state information that a model might track:
    - hitPoints, name, type, position, orientation, size, status
  - Examples of methods that a model might implement:
    - die(), getHit(), updateAnim(), insertIntoWorldDatabase(), moveTo()
- **Implementation.** Many Model implementations are polymorphic.
  - Example: CModel extends a CBaseModel class.
  - Many models can share a single rendering representation, see Type Database

Model (2)

Model Database

- **Also Known As.** World Database, World, Level
- **Intent.** Aggregate the Model objects into one database.

- **Motivation.** Collecting the models into one list simplifies several important systems.
  - The inter-object references and the "death problem". (See Controller.)
  - Some games may have more than one kind of Model Database simultaneously (i.e. TerrainModelDatabase, ObjectModelDatabase)

- **Implementation.** Some games may implement the Model Database as a simple array of Model instance pointers. Other games may choose to implement sophisticated memory management or caching solutions

- The world database is often indexed to increase search speeds.

Model (3)

Type Database

- **Intent.** Store information which is common to Model types.
- **Motivation.** There is often a great deal of common information concerning types of objects. To avoid duplication, and to simplify editing, these are separated into a database.
- **Implementation.** A Type Database is conceptually static data associated with a model sub-class.
  - Prototype state; e.g. max hit points, strength, range, cost, etc.
  - Artwork; e.g. meshes, texture-map, sprites.
  - Appearance maps. (See Appearance Map)
- Example: a Model can access a Type within the Type Database for its rendering representation

View (1)

View

- **Also Known As.** Renderer, Painter, Viewer, Interface
- **Intent.** Render the visible Models given a point of view (POV)

- **Motivation.** Renderers are often the most custom part of any game; they often define the game's technology.

- **Implementation.** The View reads the Model Database via a Spatial Index but does not modify either. Thus, typically:
  - Model and Model Database are read-only by View.
  - View is invisible to Model and Model Database.

- Many View implementations translate a Model "state" into an "appearance"
  - Example: a Model "orc1" is de-referenced and is found to be type==ORC_TYPE and frame==10. The View then finds an artwork pointer via type/frame and draws.

View (2)

Render Delegation

- **Also Known As.** Overloaded draw
- **Intent.** Pass-off special render cases to Model code.
- **Motivation.** Generic View code often becomes clotted with special cases. Render Delegation moves special cases from View into Model code.
- **Implementation.** An example clot in View code:

```
if (typeToDraw==DARTH_VADERS_SHIP)
  drawSpecialShieldEffect();
```

  - To encapsulate these kinds of special cases, the View delegates the draw back to the Model. For example: `objectToDraw->draw(x,y)`
  - The View may choose to delegate only in certain special cases, often based on type data. For example:

```
if (getType(type)->delegateDraw) object->draw(x,y);
 else drawSprite(getType(type)->sprite[frame], x, y);
```

- One major drawback of Render Delegation is that the Model code must include all of the render interface, which may be substantial.

View (3)

Appearance Map

- **Also Known As.** State to Appearance Translation, Frame Mapping
- **Intent.** Isolate Model state from Model appearance to minimize impact on controllers when art changes.
- **Motivation.** It is common for Controllers to change the appearance of the Model, especially in animation controllers. Since art may change frequently it makes sense to separate the state from the appearance.
- **Implementation.** Without an appearance map, a controller is likely to change the "frame" of an animation directly. For example:

```
if (state == WALKING) {
        model.frame = WALK_START_FRAME +
        WALK_NUM_FRAMES * (WALK_DURATION / dt) ;
}
```

In this case, if the animation is changed, the three constants WALK_XXX need to be updated and the game recompiled for the change to take effect.

- An appearance map eliminates these constants and replaces them with a lookup.
- Typically, a table is loaded at game initialize time which encodes the translation from state and delta time ("state") to frame ("appearance").

Controller (1)

Controller

- **Also Known As.** Process, Mini-process
- **Intent.** Update a Model's state based on circumstance
- **Motivation.** Controllers implement the rules of a game.
  - They determine how objects behave given a circumstance, and isolate these rules from the objects (Models) themselves.

- **Implementation.** Controllers relate to Models and Views as follows:
  - Models are read-writeable by Controllers.
  - Controllers are created and destroyed by Models, but are otherwise invisible.
  - Controllers are notified by the View (i.e. `GetInput()`)
  - Controllers are often associated with only one Model instance.
    - For example: animation, AI, pathfinding. In these cases the controller instance is usually created and destroyed synchronously with the associated model.

Controller (2)

Controller

- Some Controllers inherently have more than one associated Model.
  - Example: multi-body physics, target tracking (heat seeking missiles, etc). These controllers often maintain Model references which must be notified / garbage collected when the referenced object dies.
- Controllers are often implemented as "processes" (See Mini-kernel) but may also be implemented as "hard wired updates" in the main loop, especially for large multi-model controllers like physics.
- Some simple Controllers are stateless.
  - For example, a homing missile controller may just compute the direction to the target and apply force as necessary. Most controllers, however, are state-aware.
- Controllers should be aware of their per frame time budget
- State-aware Controllers often become significantly complicated with large switch statements.

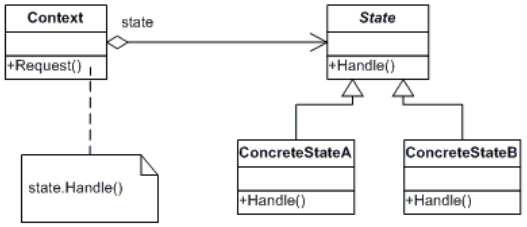
Controller (3)

Controller State Machine

- **Intent**. Track a complicated state process with a Controller.
- **Motivation.** Controllers = complicated state machines, incl. state transitions in response to events. Animation is the canonical example.
- **Implementation.** A Controller subclass with list of all state variables.
  - Example: an animation might have: currectFrame, currentAnim, lastFrameTime, etc. The process of the controller contains a switch on some primary state.

```
void Animation::doProcess() {
          switch( animState ) {
          case RUNNING_STARTING:
          case RUNNING:
          case RUNNING_STOPPING: ...
```

- Each state updates and checks for transition conditions.
  - Example: RUNNING may check to see if it is at the end of the cycle, if so, restart it.
- Some states need *double buffering* (i.e. physics simulations)
- State machines can become very complicated and difficult to maintain using this technique.



Controller (4)

Controller State Machine

- **A possible solution to „switch/case“**
  - Use a **State** base class, and overide the Controller `Update(..)` method depending on state of controller

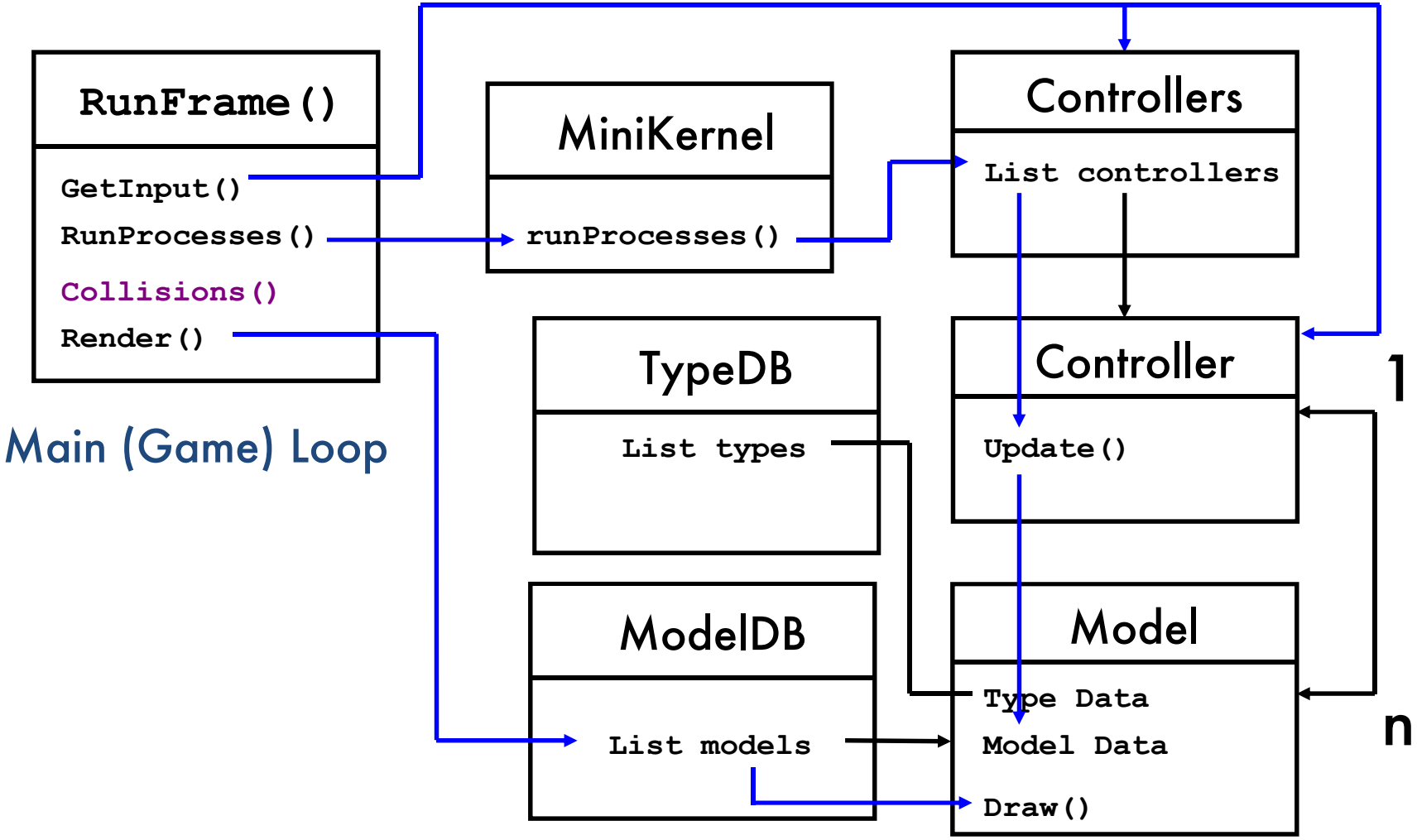
```
void CSphereController::Update(GameTime gameTime) {
    state->Update(gameTime);
}
```

  - State must check for state transition at the end of `Update(...)`

```
void CIntersectedState::Update(GameTime gameTime) {
   ....
   if (thisModel->isNotIntersected()) {
       controller->state = new NonIntersectedState()
   }
```

Model View Controller

Subset of a possible prototype



Code Example

- My prototype implementation of Game Design Patterns (C++ code)

  `http://gamedev.nealen.net/intern/docs/gamearch.zip`

- Step through the code
  - Start at

    `void CGameEngine::Init();`

    `bool CGameEngine::RunFrame();`
  - Move to View

    `void CModel::Draw();`

Rest of Today

- Game Prototype #4
- Next week: last prototype... After that, form teams
  - 2-4 people, probably 3-4 teams