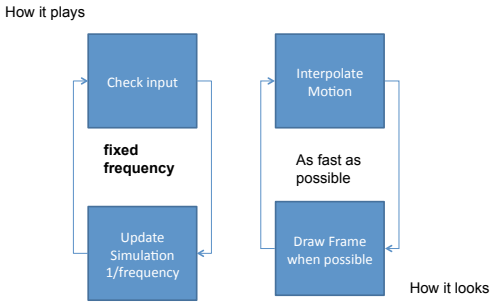
G54GAM - Games

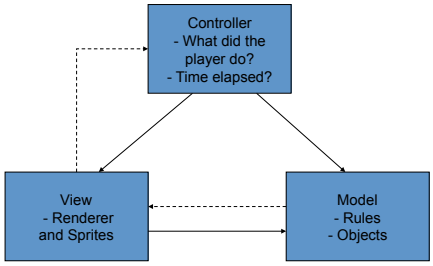
- Software architecture of a game
- Encapsulation and Autonomy

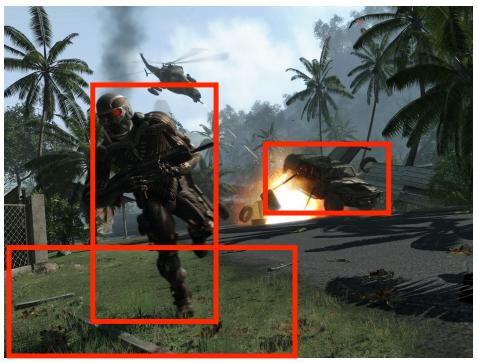
Time and "The Game Loop"

- The "heart beat" of a game
- Performs a series of tasks every **frame**
  - Game state changes over time
  - Each frame is a snapshot of the evolving game state
  - A series of frames are perceived as movement
    - E.g. 60 frames per second
- Run as fast as we can?
  - A smooth game-play experience

Decoupling

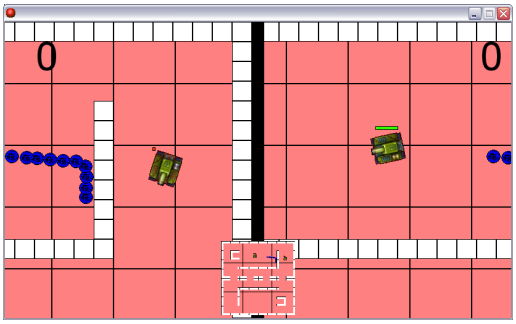






Why MVC is popular / good

- Nice modular design
  - Decouple game design from renderer
- Game world logic is bundled in the model
- Changes to the renderer / graphics do not affect the rest of the game
- Easily supports different input controllers and/or bots and AI
- Helpful when we think about networked games

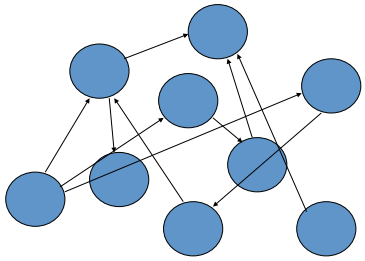


Structuring the Model

- Model
  - Objects
  - Rules
  - Together create game world and state
- Objects need to communicate with one another
- Objects need to obey rules and procedures
- Objects need to be able to do things by themselves
- How do we structure this sensibly?

Direct Communication

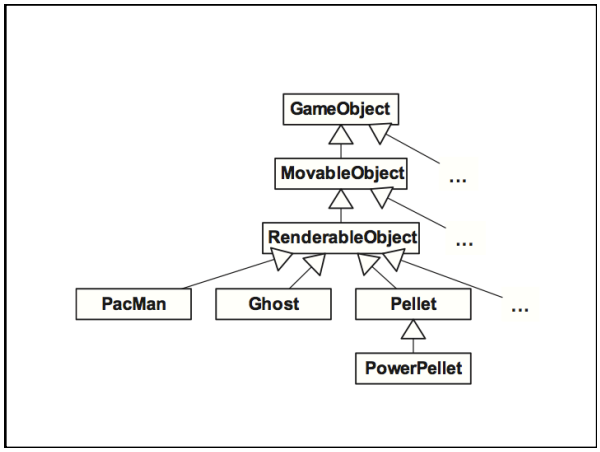
- Object A attempts to pick up object B
  - Who checks if B can be picked up
  - Which functions A must call in B to reflect pick-up
  - Many conditional statements
- Bullet hits player
  - Who decides what happens?
  - The player is damaged
  - The bullet is destroyed
  - Where do we put the logic?
- Why is this a poor design choice?
  - Exponential complexity
  - Every object needs to know how to interact with every other object
  - Very time consuming to add new objects

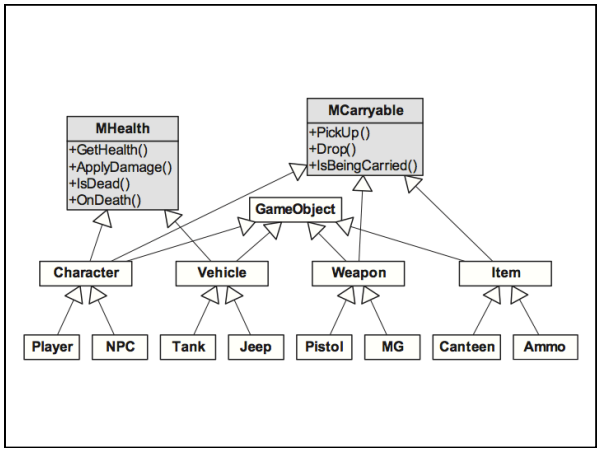


Encapsulation and Inheritance

- Generalise the kinds of functions that objects must respond to, and encapsulate this functionality within the object
  - Build an object hierarchy
  - Objects implement **interfaces**
  - Objects look after themselves – adding new objects is trivial
- Functions / Interfaces
  - Update() - Calculate where I am now
  - Render() - How I am drawn on screen
- Subsystems **iterate** object collections, but are dumb!

```
for(Object o : all the objects)
{
    o.update();
    o.render();
}
```





Doom 3

```
class idEntity : public idClass {
public:
   virtual renderEntity_t *GetRenderEntity( void );
   // run the physics for this entity
   bool RunPhysics( void );
```

Doom 3

```
class idMoveable : public idEntity {
public:
      virtual bool Collide( const trace_t
            &collision, const idVec3 &velocity );
```

Quake 3

```
#define RESPAWN_ARMOR 25
#define RESPAWN_HEALTH 35
#define RESPAWN_AMMO 40
#define RESPAWN_HOLDABLE 60
#define RESPAWN_MEGAHEALTH 35//120
#define RESPAWN_POWERUP 120
int Pickup_Powerup( gentity_t *ent, gentity_t *other ) {
   …
```

Code or data?

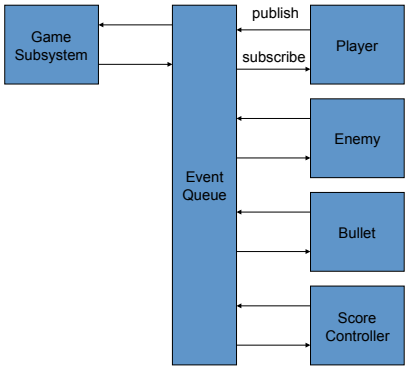
- Easy (!) to create hierarchies in code
  - Object orientation, interfaces
  - Some language specific pitfalls
    - The “diamond of death”
- A game engine should be as generic as possible
  - The code supports a certain **type** of game
- Separate specific functionality as data
  - **Data-driven** approach
  - Scripts, models, levels
  - Support DLC (Downloadable content)

Events

- Games are inherently **event** driven
  - Anything of interest that happens
    - Explosion goes off, coin being picked up
  - Need a mechanism to…
    - Inform relevant objects that something has happened
    - Respond (**handle**) the event in some way
- "Event subsystem"
  - Often provided by a game engine to make our lives easier

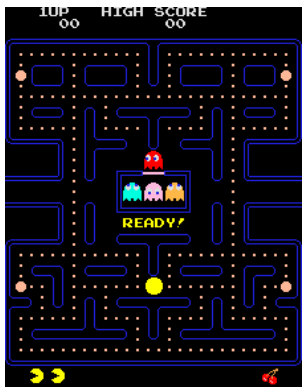
Events

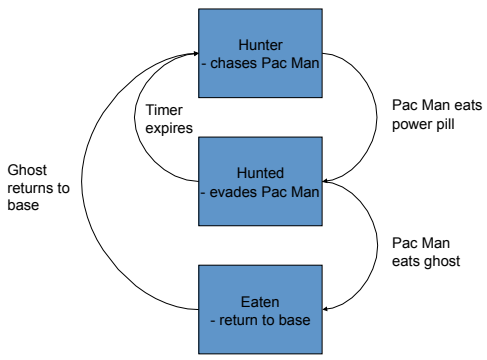
- Physics simulation
  - A bullet has collided with the player
  - Could tell all interested objects directly?
    - Bullet, player, scoreboard, health, walls, floor, sky…
- Broadcast an **event**
  - Objects **listen** for particular events to **handle**
  - Ignore events we don't care about
    - Scenery doesn't care about the score
  - Makes it easy to add new events
  - Objects can prioritise events in different ways
  - Dynamically register and unregister interest in events
    - Changing context



Finite State Machines

- Events allow objects and subsystems to talk to one another
- FSMs allow objects to be autonomous
  - Evolve from "exploding" to "destroyed"
  - Artificial Intelligence / "bots"
- Literally
  - A **finite** number of **states**
  - **Actions** allow the object to **transition** between **states**





FSM code

```
update()
{
    switch(state)
    {
      case hunting:
         chasePacMan();
      case hunted:
         evadePacMan();
       case eaten:
         returnToBase();
    }
}
```

```
handleEvent(event e)
{
    {
      case eatenPill:
         setState(hunted);
      case reachedBase:
         setState(hunting);

      …
    }
}
```

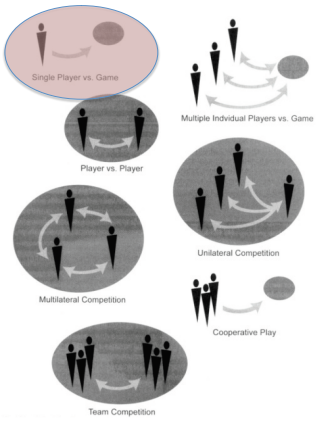
References

Look at real game code:
https://github.com/id-Software/

Multiplayer Games

- Single Player
  - Pre-defined challenges
  - Artificial Intelligence controlled opponents using Finite State Machines
  - Simple MVC design
- Multi-Player
  - More than one player can play in the same environment at the same time
  - Interaction with other players forms a key challenge of the game play
  - How can we build such a system?
  - What are some of the issues that arise?

Multiplayer / Networked Games



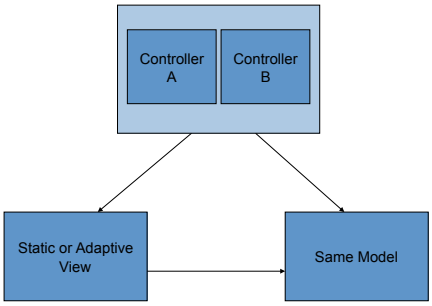
Where is the view?

- Local
  - Players are co-located
  - Share the same console / screen / pc
  - Share or split screen into two or four sections
  - Arcade games, racing, fighting, co-operative shooters
- Networked / Online
  - Players are physically separated
  - Game play is shared over the network / Internet
  - Many combinations of players 2 -> ??
  - FPSs, MMORPGs

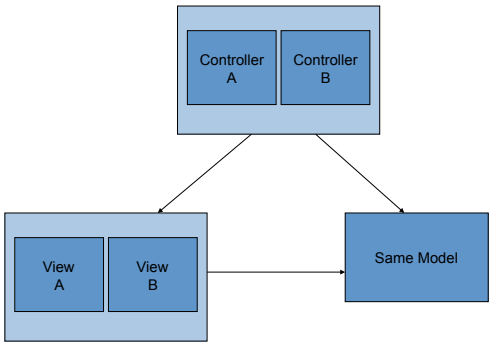
Static Shared View – Bomber Man

Adaptive Shared View – Street Fighter

Adaptive Shared View – Street Fighter



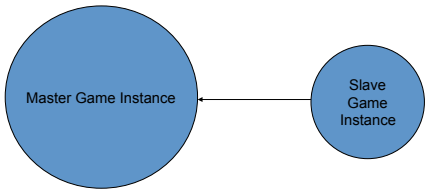
Split-Screen View – Mario Kart

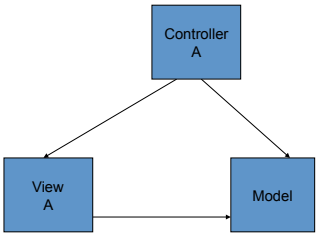


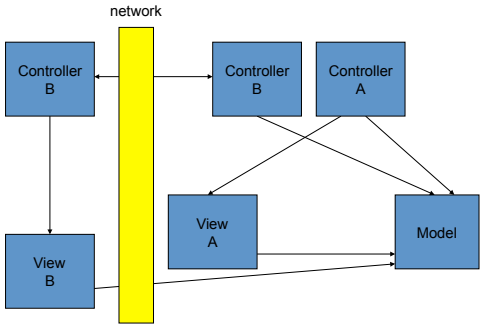
Networked / Online Game Play

- Players are physically separate
- Where is the game?
- **Master** and **slave**
  - Usually two players, local network
- **Dedicated server** and **Clients**
  - Multiple players, local network, internet
- **(Peer-to-peer)**
  - Largely theoretical

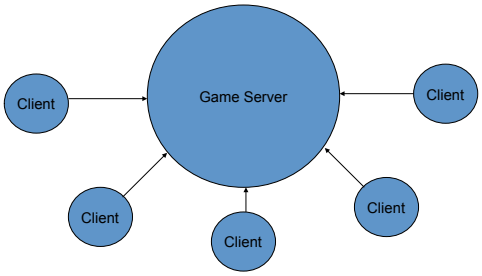
Master and Slave

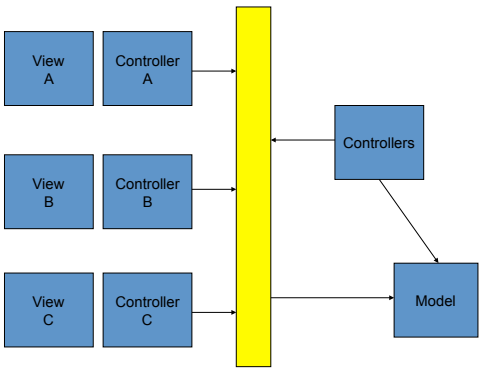






Client Server

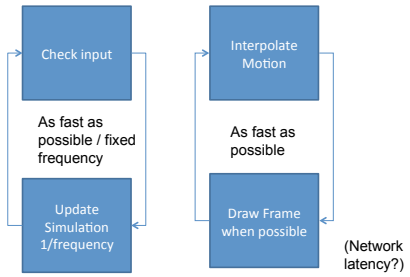




Sadly it's not that simple

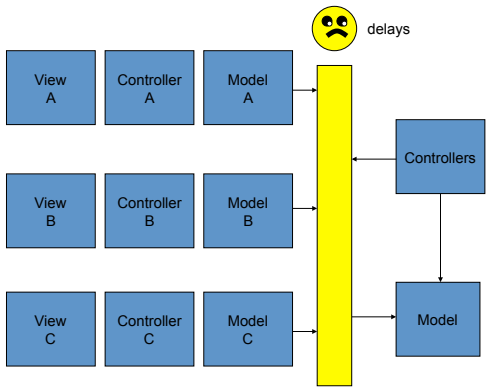
- Game loop runs at 30 Hz
- Renderer redraws at 50-100 Hz
- Each input has to…
  - Travel from the client to the server
  - Be processed by the server
  - Wait for the server loop to update the model
  - Travel from the server to the client
  - Be drawn onscreen
- For several players on the Internet playing a fast paced game
  - This is very slow
  - Requires a lot of bandwidth (each client needs to know about the current state of the model every loop)
  - Packets get lost or delayed or arrive out of order

Decoupling



Local Replication

- Give each client its own replica of the model to use
- The server is the authority on what is happening
- The client
  - Receives slow regular **snapshots** of the server model
    - UDP (fast, but unreliable @~20Hz)
  - Receives fast **deltas** – updates to the snapshot while waiting for next snapshot
  - Has a local replica of the server's logic
  - Between snapshots uses its local replica of the model to calculate the current state to render the view
  - Tells the server what it is doing, so the server can update the master model



Lag

- Network delays lead to logical Inconsistencies
- A player shoots at another player
  - The player/client thinks they hit
  - The **fire** command takes some time to get to the server
  - The server thinks the player missed
- Two players try to pick up the same gold
  - We both arrived at the gold at the same time
  - Who picked up the gold?
  - We both did, we both saw that we did
  - Who gets to keep it?

Lag

- Try to keep client and server models in sync
  - Fast (otherwise it feels slow and jerky)
  - Synchronised (to avoid logical inconsistencies)
  - Not use too much bandwidth (remember dial-up)
  - Cope with packet loss
  - Recovery (Rewind to last **snapshot)**
- Client prediction
- Entity interpolation
- Lag compensation

Entity Interpolation

- Client is responsible for frame-to-frame movement simulation and rendering
  - X is moving at speed Y with direction Z, so move it a bit
  - 100Hz
- Server is responsible for the bigger picture, the context and the consequences
  - X has picked up the gold, Y hasn't
  - X has fired a gun and hit Y
  - X has pressed UP so start them moving
  - 20Hz

Lag Compensation

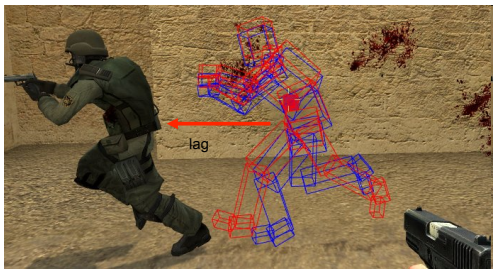
- Dead Reckoning
  - X is moving at speed Y
  - We haven't received a snapshot but need to draw a frame, so move it a bit ourselves
  - It's likely that X will continue to move at speed Y because there's nothing near it
  - …allows smoother graphics than the network allows
- Deltas
  - Only tell the client what has changed, rather than send the full model snapshot
  - Send full snapshot after a network delay so that the client gets back in sync
  - …reduces bandwidth use

Client Prediction

- The player pressed up
- Send the command to the server controller
- While we're waiting for the updated snapshot, start moving anyway
- Rewind if we got it wrong
  - Unusual, unless we're "lagging"
- …the game feels more responsive than the network allows

Server-side Lag Compensation

- The server keeps a record of current round-trip-time for all the clients (time for a packet to travel from client->server->client)
- Remember where everything was up to a second ago
- When a new command is received, estimate when it was **sent** rather than when it was **received**
- Use a historical snapshot of the model to work out what happened
- …the command is executed hopefully as if there was no lag



References

- http://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking
- Next lecture…
  - http://www.youtube.com/watch?v=0OiamBxxoXA
    - What is happening in this video?
    - Contains bad language!