CS 480/680: GAME ENGINE PROGRAMMING

THE GAME LOOP

1/17/2013
Santiago Ontañón
santi@cs.drexel.edu
Outline

- Student Presentations
- Game State
- Basic Game Loop
- Game Flow
- Resource Managers
- Another Game Loop Example (JavaScript)
- Project Discussion
Outline

- Student Presentations
- Game State
- Basic Game Loop
- Game Flow
- Resource Managers
- Another Game Loop Example (JavaScript)
- Project Discussion
Student Presentations

- Anyone has not sent me their preferences?
  - Remember that you can also propose papers other than the ones I uploaded.
  - Another option (but tell me quickly) is if any of you wants to study the source code of an existing game engine (Ogre, Doom 3, etc.) and present it in class.
Student Presentations

• Mike Kopack: “Tactical Path Planning with A*”

• Alberto Uriarte: “Navigation in a game engine”
  • “Reciprocal Collision Avoidance and Navigation for Video Games”
Outline

- Student Presentations
- **Game State**
- Basic Game Loop
- Game Flow
- Resource Managers
- Another Game Loop Example (JavaScript)
- Project Discussion
Game State

- **Game State**: set of variables and data structures that together capture the current state of the game:
  - Map state (anything that is dynamic)
  - Objects (physical state)
  - Characters (physical state, AI state)
  - Player (physical state, inventory, stats, etc.)
  - (Camera)
- Things that are NOT part of the game state: high scores, configuration (resolution, keyboard setup, etc.), pause state, etc.

- Rendering engine takes the game state and renders it to screen
- Other of modules in the game engine (physics, AI) update game state
- It is very important that the game state is clearly identified
Game State

• If the “game state” is properly identified, things like saving/loading game or synchronizing an online game are easier (some even trivial):
  • Save game: dump game state to disk
  • Load game: reload game state from disk

• If game state is distributed between many classes and modules, and not clearly delimited, this might be complicated.
Game Engine Architecture

Game Engine Functionalities
Resource Management
Utility Layer
Platform Independence Layer
SDKs
OS
DRIVERS
HARDWARE

Dependencies
Game Engine Architecture

- Where is the “Game State”?
Centralized Game State

- Neat option (might be hard to pull off in some game engines): completely separate game state
Centralized Game State

- Neat option (might be hard to pull off in some game engines): completely separate game state

Some data structure, or database with the complete game state (everything!) centralized. Other modules just read it or update it. There is NO internal state in the other modules.
Centralized Game State

- Code-wise, this means, for example:

**Game State Data Structure:**

```cpp
Class GameState {
    2DMap *map;
    List<2DBody> objects;
    2DBody *player;
    2DVector camera;
}

Class 2DBody {
    2DShape *shape;
    2DVector position;
    float rotation;
    2DVector velocity;
    float angular_velocity;
    AIState *aiState;
    AIScript *ai;
}
```

**Example, Adding an Enemy:**

```cpp
2DBoody b = new 2DRectangle();
b->setAnims(rm->getAnim("Goblin"));
b->setAI(rm->getScript("GoblinAI"));
gameState->addObject(b);
```
Centralized Game State

• Code-wise, this means, for example:

Game State Data Structure:

```cpp
class GameState {
    2DMap *map;
    list<2DBody> objects;
    2DBody *player;
    2DVector camera;
}
```

Class 2DBody {
    2DShape *shape;
    2DVector position;
    float rotation;
    2DVector velocity;
    float angular_velocity;
    AIState *aiState;
    AIScript *ai;
}

Example, Adding an Enemy:

```cpp```
2DBoody b = new 2DRectangle();
b->setAnims(rm->getAnim("Goblin"));
b->setAI(rm->getScript("GoblinAI"));
gameState->addObject(b);
```

If you need things like “hit points”, those are variables of the script, and thus, are stored in the aiState.

Notice that there is no additional code, nor state for storing the the Goblin’s state. It simply states that a “goblin” will behave as a “2DRectangle” (the physics engine will know what that means), and that the animations (graphics) are the ones identified by the “Goblin” tag in the resource manager. Also the behavior is determined by the AI Script “GoblinAI”.
Centralized Game State

• Code-wise, this means, for example:

```c++
Class GameState {
    2DMap *map;
    List<2DBody> objects;
    2DBody *player;
    2DVector camera;
}

Class 2DBody {
    2DShape *shape;
    2DVector position;
    float rotation;
    2DVector velocity;
    float angular_velocity;
    AIState *aiState;
    AIScript *ai;
}

Example, Adding an Enemy:

2DBoody b = new 2DRectangle();
b->setAnims(rm->getAnim("Goblin"));
b->setAI(rm->getScript("GoblinAI"));
gameState->addObject(b);
```

This is the “game engine” philosophy:
When planning the game, we do NOT think:
- There will be a “player class”, a “bullet class”, an “enemy class”.
What we think is:
- There will be objects that will behave like static obstacles, others can walk, others will be projectiles, etc.
Then, we use those generic objects, to create players, bullets, etc.
Distributed Game State

- Realistic option (most typical): game state is distributed between several modules
Distributed Game State

• Code-wise, this means, for example:

Example, Enemy Code:

```cpp
Class Goblin {
    2DBody *shape;
    AIScript*ai;
    int hitpoints;
    ...
}
```

Distributed state

Example, 2DBody Code:

```cpp
Class 2DBody {
    2DShape *shape;
    2DVector position;
    float rotation;
    2DVector velocity;
    float angular_velocity;
    ...
}
```
Distributed Game State

- Code-wise, this means, for example:

Example, Enemy Code:

```cpp
Class Goblin {
    2DBody *shape;
    AIScript *ai;
    int hitpoints;
    ...
}
```

Example, 2DBody Code:

```cpp
Class 2DBody {
    2DShape *shape;
    2DVector position;
    float rotation;
    2DVector velocity;
    float angular_velocity;
    ...
}
```

This option is much more natural and efficient. But makes the distinction between “game state” and the rest of the game engine blurry.

The programmer must always remember where all the variables that define the game state are. So that if the game state needs to be synchronized for a networked game, or if we want to save/load the game, everything runs fine.
Game State

• In a Game Engine:
  • Ideally separated from the rest of the code
  • Makes things like “having a separate networking module” much easier (since it only has to deal with a centralized game state)
  • Distributed Game State is easier to code (but then might give bigger headaches for networking or game state saving/loading)
  • A distributed Game State requires a very nice design of the game classes, so that they offer a nice interface to the networking code, for example, in order to update/read the state.

• In a Game:
  • It doesn’t really matter. Since the code is specific for the game, and will most likely not be reused.
Outline

- Student Presentations
- Game State
- **Basic Game Loop**
- Game Flow
- Resource Managers
- Another Game Loop Example (JavaScript)
- Project Discussion
The Game Loop

- Responsible for controlling the time-flow of the game.

- Responsible for integrating and servicing all the modules in the game engine.

- Simple game loop:
  - Repeat 50 times per second:
    - Get user input
    - Update game state
    - Render graphics
Game Engine Architecture

- Game Specific
- Game Engine Functionalities
- Resource Management
- Utility Layer
- Platform Independence Layer
- SDKs
- OS
- DRIVERS
- HARDWARE
Game Engine Architecture

- Most complex layer of the game engine, composed of multiple sub-systems
The Simplest Game Loop

G = new GameEngine();
While(!quit) {
    I = getInput(&quit);
    G.updateGameState(I);
    G.render();
    FPScontrol();
}
Delete G;
High-Level Idea of a Game Loop

G = new GameEngine();
While(!quit) {
    I = getInput(&quit);
    G.updateGameState(I);
    G.render();
    FPScontrol();
}
Delete G;
A More Realistic Game Loop

G = new GameEngine();

While(!quit) {
    pollForOSMessages();
    I = getInput(&quit);
    if (timeForUpdatingAI()) G.updateAI(&I);
    if (timeForUpdatingPhysics()) G.updatePhysics(I);
    updateStatistics();
    if (timeForRendering()) G.render();
    FPSControl();
}

Delete G;
A More Realistic Game Loop

G = new GameEngine();
While(!quit) {
    pollForOSMessages();
    I = getInput(&quit);
    if (timeForUpdatingAI()) G.updateAI(I);
    if (timeForUpdatingPhysics()) G.updatePhysics(I);
    updateStatistics();
    if (timeForRendering()) G.render();
    FPScontrol();
}
Delete G;

Play nice to the OS, and make sure the game quits if the “close window” button is pressed, etc. Also, in some systems, this is the way to get updates on mouse position, keyboard state, etc.
A More Realistic Game Loop

G = new GameEngine();
While(!quit) {
    pollForOSMessages();
    I = getInput(&quit);
    if (timeForUpdatingAI()) G.updateAI(I);
    if (timeForUpdatingPhysics()) G.updatePhysics(I);
    updateStatistics();
    if (timeForRendering()) G.render();
    FPScontrol();
}
Delete G;

Check the keyboard/mouse/touch surface state and identify any user input. Also, if there is player over the network, get their inputs.
A More Realistic Game Loop

G = new GameEngine();
While(!quit) {
    pollForOSMessages();
    I = getInput(&quit);
    if (timeForUpdatingAI()) G.updateAI(&I);
    if (timeForUpdatingPhysics()) G.updatePhysics(I);
    updateStatistics();
    if (timeForRendering()) G.render();
    FPScontrol();
}
Delete G;

Typically once or twice per second.

Update the AI of any NPC in the game.
A More Realistic Game Loop

G = new GameEngine();
While(!quit) {
    pollForOSMessages();
    I = getInput(&quit);
    if (timeForUpdatingAI()) G.updateAI(&I);
    if (timeForUpdatingPhysics()) G.updatePhysics(I);
    updateStatistics();
    if (timeForRendering()) G.render();
    FPScontrol();
}
Delete G;

With the given player inputs, run one frame of the physics simulation to move the characters and objects in the game. As often as possible.
A More Realistic Game Loop

G = new GameEngine();
While(!quit) {
    pollForOSMessages();
    I = getInput(&quit);
    if (timeForUpdatingAI()) G.updateAI(&I);
    if (timeForUpdatingPhysics()) G.updatePhysics(I);
    updateStatistics();
    if (timeForRendering()) G.render();
    FPScontrol();
}
Delete G;

Get information from the game engine to update the player score, health, etc. (since typically the HUD is not rendered by the game engine, but separately)
A More Realistic Game Loop

G = new GameEngine();
While(!quit) {
    pollForOSMessages();
    I = getInput(&quit);
    if (timeForUpdatingAI()) G.updateAI(&I);
    if (timeForUpdatingPhysics()) G.updatePhysics(I);
    updateStatistics();
    if (timeForRendering()) G.render();
    FPScontrol();
}
Delete G;

Redraw the game

Keep at 30 or 60fps (as desired)
Time Management

- Game Loop is responsible for maintaining the game running at the desired speed.
- There’s different ways to achieve that.
- Old fashioned way (**do not use this!**):

  ```
  While(!quite) {
    I = getInput(&quit);
    G.updateGameState(I);
    G.render();
    delay(10ms);
  }
  ```

  Insert a pause (e.g. 10 milliseconds), so that the game loop runs at the desired speed.
int time = SDL_GetTicks();
int REDRAWING_PERIOD = 20;
bool need_to_redraw = true;

while (!quit) {
    int act_time = SDL_GetTicks();
    if (act_time - time >= REDRAWING_PERIOD) {
        time += REDRAWING_PERIOD;

        keyboard->cycle();
        if (!game->cycle(k)) quit = true;
        need_to_redraw = true;
    }

    if (need_to_redraw) {
        game->draw();
        need_to_redraw = false;
    }

    SDL_Delay(1);
}
int time = SDL_GetTicks();
int REDRAWING_PERIOD = 20;
bool need_to_redraw = true;

while (!quit) {
    int act_time=SDL_GetTicks();
    if (act_time-time>=REDRAWING_PERIOD) {
        time+=REDRAWING_PERIOD;
        keyboard->cycle();
        if (!game->cycle(k)) quit=true;
        need_to_redraw=true;
    }

    if (need_to_redraw) {
        game->draw();
        need_to_redraw=false;
    }

    SDL_Delay(1);
}
Time Management (Better C++ Example)

```cpp
int time = SDL_GetTicks();
int REDRAWING_PERIOD = 20;
int MAX_FRAME_SKIP = 10;
bool need_to_redraw = true;

while (!quit) {
    int act_time = SDL_GetTicks();
    int frames = 0;
    while (act_time - time >= REDRAWING_PERIOD && frames < MAX_FRAME_SKIP) {
        time += REDRAWING_PERIOD;
        keyboard->cycle();
        if (!game->cycle(k)) quit = true;
        act_time = SDL_GetTicks();
        need_to_redraw = true;
        frames++;
    }
    if (time < act_time) time = act_time;

    if (need_to_redraw) {
        game->draw();
        need_to_redraw = false;
    }

    SDL_Delay(1);
}
```

If the computer is too slow to run at the desired FPS, we can try to execute more than one update per redrawing frame ("skip frames")

If after skipping the maximum number of frames, we haven’t yet caught up. It might be that the game lost focus or something. So, don’t try to catch up.
Time Management (Better C++ Example)

int time = SDL_GetTicks();
int REDRAWING_PERIOD = 20;
int MAX_FRAME_SKIP = 10;
bool need_to_redraw = true;

while (!quit) {
    // insert here the event pump …
    int act_time=SDL_GetTicks();
    int frames = 0;
    while( act_time - time >= REDRAWING_PERIOD && frames < MAX_FRAME_SKIP )
        time+=REDRAWING_PERIOD;

        keyboard->cycle();
        if (!game->cycle(k)) quit=true;
        act_time=SDL_GetTicks();
        need_to_redraw=true;
        frames++;
    }
    if (time < act_time) time = act_time;

    if (need_to_redraw) {
        game->draw();
        need_to_redraw=false;
    }

    SDL_Delay(1);
}
Time Management (Better C++ Example)

```cpp
int time = SDL_GetTicks();
int REDRAWING_PERIOD = 20;
int MAX_FRAME_SKIP = 10;
bool need_to_redraw = true;

while (!quit) {
    while (SDL_PollEvent( &event ) ) {
        switch ( event.type ) {
            case SDL_KEYDOWN:
                if (event.key.keysym.sym == SDLK_F12) {
                    quit = true;
                }
                break;

            case SDL_MOUSEBUTTONDOWN:
                game->MouseClick(event.button.x, event.button.y);
                break;

            case SDL_QUIT:
                quit = true;
                break;
        } /* switch */
    } /* while SDL_PollEvent */

    // rest of the game loop …
}
```

Example Event Pump (C++ using SDL)
Continuous Time Management

- All the game loops we’ve seen so far run the game in “cycles”, assuming the game will run at a fixed FPS.
- Another option is to update as frequently as possible (use 100% of the CPU power), and just run smoother simulations when possible:

```cpp
int lastTimeRendered = getTime();
While(!quit) {
    I = getInput(&quit);
    int time = getTime();
    int delta = time - lastTimeRendered;
    int lastTimeRendered = time;
    G.updateGameState(I, delta);
    G.render();
    FPScontrol();
}
```
Continuous Time Management

- All the game loops we’ve seen so far run the game in “cycles”, assuming the game will run at a fixed FPS.
- Another option is to update as frequently as possible (use 100% of the CPU power), and just run smoother simulations when possible:

```c
int lastTimeRendered = getTime();
While(!quit) {
    I = getInput(&quit);
    int time = getTime();
    int delta = time – lastTimeRendered;
    int lastTimeRendered = time;
    G.updateGameState(I, delta);
    G.render();
    FPScontrol();
}
```

The physics, AI code, etc. receive the amount of time for which they need to simulate.
Continuous Time Management

```cpp
While(!quit) {
    I = getInput(&quit);
    G.updateGameState(I);
    G.render();
    FPScontrol();
}
```

Example physics code:
```cpp
Void update(...) {
    ...
    position += velocity;
    rotation += angular_velocity;
}
```

```cpp
int lastTimeRendered = getTime();
While(!quit) {
    I = getInput(&quit);
    int time = getTime();
    int delta = time – lastTimeRendered;
    int lastTimeRendered = time;
    G.updateGameState(I, delta);
    G.render();
    FPScontrol();
}
```

Example physics code:
```cpp
Void update(…, float delta) {
    ...
    position += velocity * delta;
    rotation += angular_velocity * delta;
}
```
Continuous Time Management

Most commercial game engines use this variant (Quake II, Panda3D, etc.) since it maximally exploits the hardware.

However, the main problem is that it introduces non-determinism. Which is a problem for: networking and debugging.

I recommend a standard game loop for your project. But it’s ok if you want to try the continuous version.

Example physics code:

```
Void update(float delta) {
    position += velocity * delta;
    rotation += angular_velocity * delta;
}
```

```
int lastTimeRendered = getTime();
While(!quit) {
    I = getInput(&quit);
    int time = getTime();
    int delta = time – lastTimeRendered;
    int lastTimeRendered = time;
    G.updateGameState(I, delta);
    G.render();
    FPScontrol();
}
```
On Multithreaded Game Loops

- Game loops are typically single threaded
- Recent parallel hardware is pushing developers to think of multithreaded game loops:
  - Run each subsystem in a thread:
    - One thread for continuous rendering
    - One thread for continuous physics update
    - One thread for AI
    - Etc.
- They are mode complex, and need synchronization. Much more error prone, and hard to debug:
  - I do NOT recommend you going for a multithreaded game loop for your project
Networked Multiplayer Game Loops

- Basic Client-Server

**Server Side:**

```c
While(!quit) {
    I = getInputfromClient(&quit);
    G.updateGameState(I);
    FPScontrol();
}
```

**Client Side:**

```c
While(!quit) {
    I = getInput(&quit);
    sendInputToServer(I);
    getUpdateFromServer();
    G.render();
    FPScontrol();
}
```
Networked Multiplayer Game Loops

- Basic Client-Server

**Server Side:**

```c
While(!quit) {
    i = getInputFromClient(&quit);
    G.updateGameState(i);
    FPScontrol();
}
```

**Client Side:**

```c
While(!quit) {
    i = getInput(&quit);
    sendInputToServer(i);
    getUpdateFromServer();
    G.render();
    FPScontrol();
}
```
Networked Multiplayer Game Loops

• Player Prediction:
  • If you implement the game loop as in the previous slide, there will be lag (the actions executed by the player need to be sent to the server, and then the game state sent back to the client)

• Commercial game engines implement “player prediction”:
  • The client also implement the physics/AI code, and updates the game state locally (so that the player actions have immediate effect).
  • When an update is received from the server, all of the local changes are overwritten (i.e. the server version of the game state is the “master copy”).
Networked Multiplayer Game Loops

- Peer-to-peer
  - All computers act like servers and like clients
  - Each computer takes “ownership” of a subset of the objects in the game (e.g. some particular room), and acts as a “server” for those objects
  - For the rest of objects, it acts as a client, and just receives updates from the other computers in the network.

- Code is more complex than a client-server architecture (since physics engine and other sub-modules, need to handle two types of objects: owned objects, and objects owned by other computers).
Outline

• Student Presentations
• Game State
• Basic Game Loop
• Game Flow
• Resource Managers
• Another Game Loop Example (JavaScript)
• Project Discussion
Game Flow

• Refers to how the game transitions from the main menu to gameplay, or to pause, or to the game over screen, etc.

• Typically implemented outside of the game engine (but can be inside):
  • Game engine only implements the “gameplay” part of the game
  • Game-specific code implements menus, game over screen, keyboard configuration screens, etc.

• Each game does it in a different way. Following slides describe a neat way to do it using Finite State Machines. Which I recommend for your projects.
FSM-based Game Flow

- Main idea: design the game flow as a finite state machine. For example:
FSM-based Game Flow

- A game will be a two-tiered application, where the top tier implements the game flow, and the bottom tier implements the game engine (which might only be used in the “game play” FSM state).
FSM-based Game Flow

- Transball GL:
  [http://www.youtube.com/watch?v=AtKwU0hJL50&feature=youtu.be](http://www.youtube.com/watch?v=AtKwU0hJL50&feature=youtu.be)

---

### Game Loop

TransballApp class:

```c
bool cycle(Keyboard *k);
Void draw(int dx, int dy);
```

### Game Engine

```c
bool cycle(Input *I, ResourceManager *rm);
Void draw(ResourceManager *rm);
```

- Exactly as seen in the slides before
- Implements the game flow (next slides)
- Knows about keyboard configuration, pause, etc.
- Doesn’t know where the input comes from (keyboard, network, replay)
- It just implements the game
FSM-based Game Flow

• Transball GL:
  http://www.youtube.com/watch?v=AtKwU0hJL50&feature=youtu.be

Game Loop

TransballApp class:
  bool cycle(Keyboard *k);
  Void draw(int dx, int dy);

Game Engine:
  bool cycle(Input *I, ResourceManager *rm);
  Void draw(ResourceManager *rm);

Exactly as seen in the slides before

Implements the game flow (next slides)
  Knows about keyboard configuration, pause, etc.

Doesn’t know where the input comes from
  (keyboard, network, replay)
  It just implements the game
FSM-based Game Flow

```cpp
bool TGLapp::cycle(KEYBOARDSTATE *k)
{
    int old_state=m_state;

    switch(m_state) {
    case TGL_STATE_PLAYERPROFILE:m_state=playerprofile_cycle(k); break;
    case TGL_STATE_INTRO:m_state=intro_cycle(k); break;
    case TGL_STATE_MAINMENU:m_state=mainmenu_cycle(k); break;
    case TGL_STATE_CONFIGURE:m_state=configure_cycle(k); break;
    case TGL_STATE_LEVELPACKSCREEN:m_state=levelpackscreen_cycle(k); break;
    case TGL_STATE_LEVELPACKBROWSER:m_state=levelpackbrowser_cycle(k); break;
    case TGL_STATE_PREGAME:m_state=pregame_cycle(k); break;
    case TGL_STATE_GAME:m_state=game_cycle(k); break;
    case TGL_STATE_POSTGAME:m_state=postgame_cycle(k); break;
    case TGL_STATE_SAVEREPLAY: m_state=savereplay_cycle(k); break;
    case TGL_STATE_REPLAYBROWSER: m_state=replaybrowser_cycle(k); break;
    case TGL_STATE_HIGHSCORES: m_state=highscores_cycle(k); break;
    case TGL_STATE_HIGHSCORES_TIMES: m_state=highscores_times_cycle(k); break;
    case TGL_STATE_LOADLEVELPACK: m_state=loadlevelpack_cycle(k); break;
    default:return false;
    } /* switch */

    if (old_state==m_state) {
        m_state_cycle++;
    } else {
        m_state_cycle=0;
    } /* if */

    m_SFXM->next_cycle();
    m_previous_state = old_state;
    return true;
} /* TGLapp::cycle */
```
FSM-based Game Flow

- **Main idea**: design the game flow as a finite state machine. For example:

  ```cpp
  TransballApp class:
  
  bool cycle(Keyboard *k);
  Void draw(int dx, int dy);
  ```
FSM-based Game Flow

- Main idea: design the game flow as a finite state machine. For example:

```plaintext
TransballApp class:
bool cycle(Keyboard *k);
Void draw(int dx, int dy);
```

When this transition happens, a new instance of the game engine is created.
FSM-based Game Flow

- Main idea: design the game flow as a finite state machine. For example:

  ```
  TransballApp class:
  bool cycle(Keyboard *k);
  void draw(int dx, int dy);
  ```

  In this state, the keyboard is read, and translated into "input" (commands like "left", "right", "fire"), which are given to the game engine.

  That input is saved to a list, so that it can later be saved to disk for a "replay"
FSM-based Game Flow

- Main idea: design the game flow as a finite state machine. For example:

Pause is implemented simply as not calling the “updateGameState” function of the game engine.

We still call the “render” method though, so that we see the game in the screen.
FSM-based Game Flow

Main idea: design the game flow as a finite state machine.

If we want to show some clip of game-play as the background in the main menu, like in Transball, we simply instantiate a game engine, and give the data from a replay as the input. Then render the game, and we draw the menu on top of that.

TransballApp class:

bool cycle(Keyboard *k);
Void draw(int dx, int dy);

If we want to show some clip of game-play as the background in the main menu, like in Transball, we simply instantiate a game engine, and give the data from a replay as the input. Then render the game, and we draw the menu on top of that.
FSM-based Game Flow

• Very simple and organized way to encode the game flow

• Separating the game engine from the game flow is very useful:
  • We could instantiate 4 game engines at the same time, and render the 4 of them at the same time (splitting the screen). We could even pause each of them separately (just by not calling their “updateGameState” method).
  • If our game engine is deterministic, the game flow can handle networked game play without the game engine even knowing that the game is online:
    • Get the input from the remote player from the network, and give it to the game engine as “input”.
    • Then, transfer the game state to the client.
Outline

• Student Presentations
• Game State
• Basic Game Loop
• Game Flow
• Resource Managers
• Another Game Loop Example (JavaScript)
• Project Discussion
Resource Managers

- Centralizing the storage and loading of resources
- Some resources (audio, textures, video, etc.) might be very large, and should not be loaded more than once, even if they are used by multiple classes.

- Resource manager:
  - Loads resources
  - Manages lifetime of resources: unloads those that are no longer needed
  - Manages memory usage
  - Streaming (asynchronous resource loading, if the engine supports it)
Typical Resource Manager

- External data-base (e.g. xml file with properties of each resource, or just a well organized directory structure).

```java
class SoundManager() {
    Map<String, Resource> resources;
    int memoryUsed;

    public SoundManager(File database) {
        ...
    }

    Sound getSound(String ID) {
        Resource r = resources.get(ID);
        if (r==null) r = load(ID);
        return r.sound;
    }
}

class Resource() {
    int lastTimeUsed;
    int size;
    Sound sound;
}
```
Resource Managers

• A Game Engine might have a centralized resource manager (typical in large game engines)

• A series of resource managers for different data types: images, sounds, etc. (typical in small engines)

• A simple resource manager can be typically implemented in a VERY few lines of code, and can be very useful. Recommended for your engines.
Outline

• Student Presentations
• Game State
• Basic Game Loop
• Game Flow
• Resource Managers
• Another Game Loop Example (JavaScript)
• Project Discussion
Example Game Loop in JavaScript

gameRun = function() {
  var loops = 0;
  if (lastTimeMouseActive + 1000 < (new Date).getTime()) {
    mouseState.x = -1;
    mouseState.y = -1;
  }
  while ((new Date).getTime() > nextGameTick && loops < maxFrameSkip) {
    Game.update();
    for (i = 0; i < 256; i++) {
      oldKeyboardState[i] = keyboardState[i];
      if (lastKeyPressed == i && keyboardState[i] == 1) timePressed[i]++;
      else timePressed[i] = 0;
    }
    mouseState.oldButton = mouseState.button;
    nextGameTick += skipTicks;
    loops++;
  }
  if (loops >= maxFrameSkip) nextGameTick = (new Date).getTime() + skipTicks;
  if (loops > 0) {
    canvas.width = width;
    canvas.height = height;
    Game.draw();
  }
};

// Start the game loop
_intervalId = setInterval(gameRun, 1000 / FPS);
Example Game Loop in JavaScript

gameRun = function() {
    var loops = 0;
    if (lastTimeMouseActive + 1000 < (new Date).getTime()) {
        mouseState.x = -1;
        mouseState.y = -1;
    }
    while ((new Date).getTime() > nextGameTick && loops < maxFrameSkip) {
        Game.update();
        for (i = 0; i < 256; i++) {
            oldKeyboardState[i] = keyboardState[i];
            if (lastKeyPressed == i && keyboardState[i] == 1) timePressed[i]++;
            else timePressed[i] = 0;
        }
        mouseState.oldButton = mouseState.button;
        nextGameTick += skipTicks;
        loops++;
    }
    if (loops >= maxFrameSkip) nextGameTick = (new Date).getTime() + skipTicks;
    if (loops > 0) {
        canvas.width = width;
        canvas.height = height;
        Game.draw();
    }
};

// Start the game loop
_intervalId = setInterval(gameRun, 1000 / FPS);

document.onkeydown = function(evt) {
    keyboardState[evt.keyCode] = 1;
    lastKeyPressed = evt.keyCode;
    // if CTRL/ALT/COMMAND are used, pass those events to the browser:
    if (evt.keyCode == 17 ||
        evt.keyCode == 18 ||
        evt.keyCode == 91 ||
        evt.keyCode == 93 ||
        keyboardState[17] == true ||
        keyboardState[18] == true ||
        keyboardState[91] == true ||
        keyboardState[93] == true) return true;
    return false;
};

document.onkeyup = function(evt) {
    keyboardState[evt.keyCode] = 0;
    // if CTRL/ALT/COMMAND are used, pass those events to the browser:
    if (evt.keyCode == 17 ||
        evt.keyCode == 18 ||
        evt.keyCode == 91 ||
        evt.keyCode == 93 ||
        keyboardState[17] == true ||
        keyboardState[18] == true ||
        keyboardState[91] == true ||
        keyboardState[93] == true) return true;
    return false;
};
And…

- This week I couldn’t figure out an excuse for including game screenshots… you’ll have to wait for next week ;}

}
Outline

- Student Presentations
- Game State
- Basic Game Loop
- Game Flow
- Resource Managers
- Another Game Loop Example (JavaScript)
- Project Discussion
Links to Game Videos

• Two interesting games:
  • Achron: http://www.youtube.com/watch?v=GcZbpL6ueM8
  • Monaco: http://www.youtube.com/watch?v=Up7u2O5n9xg&feature=youtu.be
  • BrowserQuest: http://www.youtube.com/watch?v=kYcNJQ3Y6Sg
  • Vessel: http://www.youtube.com/watch?v=T6XqizRDX-o
• Remember that you are building a “Game Engine” and NOT a “Game”.

• When you send me your ideas:
  • Do NOT say things like “we are building a zombie game”. Zombies, or pirates, or dragons are part of games, not game engines. Game engines know about meshes vs 2D graphics, or about collision spheres vs collision boxes. But not whether those boxes represent zombies or dragons.
  • You are welcome to create a game with your engine, but things like storyline, etc. will NOT be graded.
Project Discussion

• An example:
  • A student wants to work on “physics simulation”, and the group is working on a 3D platformer (Mario Galaxy-like):
    • By working on “physics simulation”, you will have to build a physics simulation module suitable for a game engine.
    • This means defining a set of physics primitives your module will support: e.g. point-like objects, rigid bodies, ropes, etc.
    • Defining which functionalities will your physics module support: gravity, bouncing (requires collision detection), friction, hinges, etc.
    • Determining the algorithm that your module will use to simulate: numerical integration, etc.
    • Once this is built, then you use it in the game.
  • It is NOT OK (i.e. will have a low grade), if you just write code inside of the character classes, or the enemy classes, so that they behave like there is gravity, etc. Remember that this is a game engine! i.e. it should work for creating more than one game.
Project Discussion

• Another example:
  • Imagine a student that wants to work on “animation”:
  • This would be BAD:
    • Work on a FPS game where the main character is a 4-legged robot. I will focus on the animation of the legs, and make sure it looks realistic.
  • This would be GOOD:
    • Work on a FPS game engine that would allow for mechanical characters. My animation engine will allow for characters to be defined as different pieces (e.g. legs, arms, etc.) and it can play different animations on them to enable effects such as: walking, crouching, reaching for objects, etc. It would allow for animation blending, for smooth transitions between different animations.
    • As a demo, I will have a 4-legged robot with a realistic walking animation. Animation blending will be demonstrated by showing how the character can smoothly transition from walking to standing still at any point during the animation.
For Today

- I just need groups, topics, and a general idea of your direction.

- For week 4:
  - Groups
  - Topics
  - Specific algorithms
  - Specific libraries you will use
  - Specific structure of your game engine (not necessarily a class diagram, see the diagrams shown in class)
  - Demo you will create