PROGRAMMING FOR MOBILE DEVICES

Programming Mobile Devices

HTML5 Games Development

Web Game Market

- Games played in browsers are very different from the AAA titles available for game consoles
  - Cheaper – mostly 'free'
  - Can not require specialist hardware or library support
    - Since most browsers won't provide this
  - Less commitment from the user/player
    - They didn't pay £50.00 for it
    - They don't expect to spend a week learning the moves
    - Instant access (in marketing terms – low friction) = throw-away
- HTML5 has taken over from Flash, which was the standard platform for web games till quite recently

Web Game Developers

- Web Games are easier to produce
  - Simple 2-D games can be developed by an individual
  - More complex 3D games still require expensive assets (models, imagery, landscape design etc.)
    - Still cheaper than AAA titles because of platform limitations
      - Lower resolutions, hardware capabilities etc.
  - Skill-set for game development is less demanding
    - Javascript vs. C++
    - Common web framework vs. multiple complex platforms (e.g. Sony Playstation Cell processors)
  - Lower marketing costs
    - Good games can go viral very quickly
    - (Poor ones fade away just as quickly)
- Typically, web games are developed for free distribution
  - Web Content is difficult to sell (no standard pay-channels, too easy to re-distribute)
  - Brands use games as promotional devices (e.g. McDonalds, HBO, Amazon)

Web Game Frameworks

- There is an embarrassingly big number of web game frameworks
  - Some are very good – Phaser, Pixi, Construct 2, MelonJS
  - Others will never become big enough to provide the necessary level of support
    - Documentation, Tutorials, Example apps, Books, user-groups etc.
- Currently, a few of these are mobile-browser capable
  - Phaser is probably the best for JS developers
    - Developed at M.I.T, Open Source, many on-line tutorials, some books on the way, well documented, lots of examples
    - https://github.com/photonstorm/phaser
      - This is a GitHub repository
      - Look well down the page for links to documents, tutorials etc.

What's in a Game Framework?

- The point of any framework is to cover the most common use-cases
  - It does all the "normal" stuff, leaving you to add the application-specific features
- In a game framework, this means
  - Screen rendering
  - Asset management (e.g. backgrounds, sprite-sheets, audio effects, background music)
  - Game-Loop management (see later)
  - Sprites – Player and Non-player game characters
  - User-input management
  - Character Interactions – e.g. collision detection, game-events
  - Start-up and shut-down
- All of this must be provided in a way that does not over-complicate the job for game developers
  - i.e. all of these components must be easy to write game-specific code around

What's in a Game Framework?

Game Loop

- The heart of ANY game (even AAA titles)
  - A deceptively simple construct:

```
while (true) {
        getUserInputs();
        updateGameObjects();
        renderGameObjects();
}
```

  - This keeps the game "moving"
  - Using it, any game scenario can be managed
    - Multi-player games
    - Turn-by-turn
    - Continuous action games etc.

Game Objects?

- Games and OOP belong together
  - Pre OOP, game code got increasingly complex and difficult to design
  - OOP allows us to deal in game elements
    - Player character, Non-player characters, game stage (e.g. a tiled maze), enemies, game furniture (e.g. doors, pick-ups, bullets etc.)
    - Most of these elements can be developed in isolation and incorporated into a game in multiples

e.g. a Sprite

- A.K.A. game-character, player-missile graphic
  - A Sprite type can become the basis for many in-game objects
  - A typical game will involve many sprites
  - Sprites perform the basic operations in games
    - Define methods of movement
    - Take part in interactions (collisions)
    - Depict game characters on-screen
    - Represent the player (and her enemies)

```
sprite = {
        image: imageFile,
        cellCount: number_of_cells_in_image,
        currentCell: 0,
        location: [x, y],
        velocity: [dx, dy],
        direction: 0,  // a number from 0 to 3
        nextCell: nextCellFunction,
        setLocation: setLocationFunction,
        setVelocity: setVelocityFunction,
        update: updateFunction,
        draw: drawFunction
}
```

A Phaser Game Structure

- Phaser's development model is fairly simple
  - A Game object with three core functions:
    - preload() – loads assets etc.
    - create() – sets up the game – creates sprites, set up physics system, defines user-interactions
    - update() – the update() and render() stages of a game loop, combined
  - An extra render() step can be used in de-bugging
    - the function is called ***after*** the render phase

```
var game = new Phaser.Game(800, 600,
                Phaser.AUTO, 'dom-element-id',
                {
                 preload: preload,
                 create: create,
                 update: update
                }
);

function preload() {
}

function create() {
}

function update() {
}
```

Width & height of game canvas

Rendering context, Phaser.CANVAS, Phaser.WEBGL or Phaser.AUTO

ID of a DOM element to put the game into

An object that maps game functions to Javascript function definitions. In this case, the three most important core functions (there are more)

preload() function

- Typically, this is used to add assets to the game
  - Assets are files that need to be downloaded to the browser
    - preload() is a way of making sure these are in-place before the game code starts
  - Normally, PNG files to represent the game background, tiles (graphic elements that can be repeated on the screen), sprite-sheets etc.), audio files etc.
  - In an early Phaser tutorial, this looks like:

```
function preload() {
    game.load.image('sky', 'assets/sky.png');
    game.load.image('ground', 'assets/platform.png');
    game.load.image('star', 'assets/star.png');
    game.load.spritesheet('dude', 'assets/dude.png', 32, 48);
}
```

Width x height of a cell in the sprite-sheet

create() function

- Initializes application:
  - Sets up core game variables
  - Starts up physics system, configures and adds sprites, game background, animations, input controls etc.
  - Is called once after the preload() function

```
var ball;

function create() {
    game.physics.startSystem(Phaser.Physics.ARCADE);
    ball = game.add.sprite(
                        game.world.randomX, 200, 'ball');
    game.physics.arcade.enable(ball);
    game.physics.arcade.gravity.y = 200;
    ball.body.velocity.set(200, 200);
    ball.body.bounce.set(1, 1);
    ball.body.collideWorldBounds = true;
    game.input.onDown.add(moveBall, this);
}
```

update() function

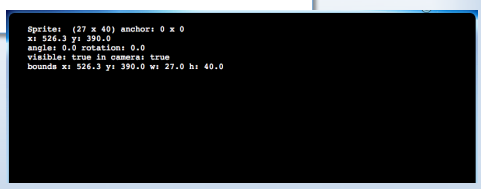
- Defines a game "step", involving game-play
  - collecting user-input
  - action of "physics" on characters
  - interactions between characters etc.
  - Updating score and score display
  - Depending on create() stage, may not need to be coded (e.g. previous slide)
- update() step ends with all game elements being rendered on screen (automatically)
- update() in Phaser can amount to all of the contents of the game-loop seen earlier (input, update, render)

```
function update() {
    game.physics.arcade.collide(sprite, sprite2);
    game.physics.arcade.collide(sprite, sprite3);
}
```

Debugging Phaser

- An additional function can form part of the game loop as a de-bugging aid
  - render() function is not normally needed
  - Can be used to put debug information directly on the game screen

```
function render() {
    game.debug.bodyInfo(sprite, 32, 32);
}
```



Sensible game structure

- Create functions to handle situations with multiple steps
  - e.g. in a collision with a pick-up (e.g. coin, power pill etc.), delete pick-up, add points, update points display
- Create objects if state is needed
  - e.g. Sprite with health and powers
    - extend the Sprite type and add these properties
    - add methods to the new type to make management easier
- Check Phaser examples at http://examples.phaser.io/
  - These tend to be single issue examples – e.g. adding gravity, rendering text, creating character animations etc.
  - The coding is simple to ease understanding, but remember that in a real game, complexity is guaranteed, and so be prepared to create types (classes) and organizations as needed

Lab Work

- Since you're probably deep in your project just now, lab work is not mandatory
- However, if you're interested…
  - There is a good Phaser tutorial at http://www.photonstorm.com/phaser/tutorial-making-your-first-phaser-game
- A suggestion: if you decide to do this, DON'T use copy & paste
  - The chocolate teapot of educational exercises