Game Development using HTML5 and JavaScript Framework

Week 8

Mario.Soflano@uws.ac.uk

Phaser Physics

The Physics Manager is responsible for looking after all of the running physics systems. Phaser supports 4 physics systems: Arcade Physics, P2 and Ninja Physics.

Game Objects (such as Sprites) can only belong to 1 physics system, but you can have multiple systems active in a single game.

For example you could have P2 managing a polygon-built terrain landscape that an vehicle drives over, while it could be firing bullets that use the faster (due to being much simpler) Arcade Physics system.

Phaser Physics

Arcade Physics is for high-speed AABB (axis-aligned bounded rectangles) collision only.

This is cheap to compute, and fast, which is probably why they recommend it for high-speed collisions.

Simplest physics: it handles objects without rotation and only checks if the image (which is a rectangle) overlaps with another image

One issue with AABB is that it doesn't guarantee that there really is a collision; you may have a completely transparent area overlapping.

Physics – Arcade Physics - Initiation

```
//Initiate arcade physics
    game.physics.startSystem(Phaser.Physics.ARCADE);

//Enable Arcade Physics for the sprite
game.physics.enable(sprite, Phaser.Physics.ARCADE);
game.physics.enable([knocker,ball], Phaser.Physics.ARCADE);
game.physics.arcade.enable([sprite, sprite2]);

spriteName.enableBody = true;
spriteName.physicsBodyType = Phaser.Physics.ARCADE;
```

Physics – Arcade Physics - Methods

- moveToPointer(displayObject, speed, pointer, maxTime) - Move the given display object towards the pointer at a steady velocity

DisplayObject: The display object to move

Speed: The speed it will move, in pixels per second (default is 60 pixels/sec)

Pointer: The pointer to move towards. Defaults to Phaser.Input.activePointer.

MaxTime: Time given in milliseconds (1000 = 1 sec). If set the speed is adjusted so the object will arrive at destination in the given number of ms.

spriteName.rotation = game.physics.arcade.moveToPointer(sprite, 60, game.input.activePointer, 500);

Physics – Arcade Physics - Methods

- angleBetween(source, target): Find the angle in radians between two display objects (like Sprites)

arrow.rotation = game.physics.arcade.angleBetween(arrow, target);

- angleToPointer(displayObject, pointer): Find the angle in radians between a display object (like a Sprite) and a Pointer, taking their x/y and center into account.

sprite.rotation = game.physics.arcade.angleToPointer(sprite);

- distanceToPointer(displayObject, pointer): Find the distance between a display object (like a Sprite) and a Pointer.

analog.height = game.physics.arcade.distanceToPointer(arrow);

Physics – Arcade Physics - Methods

//normally in update state

collide(object1, object2, collideCallback, processCallback, callbackContext): Checks for collision between two game objects. You can perform Sprite vs. Sprite, Sprite vs. Group, Group vs. Group, Sprite vs. Tilemap Layer or Group vs. Tilemap Layer collisions. Both the first and second parameter can be arrays of objects, of differing types.

CollideCallback: An optional callback function that is called if the objects collide

ProcessCallback: A optional callback function that lets you perform additional checks against the two objects if they overlap.

CallbackContext: The context in which to run the callbacks.

```
game.physics.arcade.collide(sprite1, sprite2, collisionHandler, null, this);
```

Physics – Arcade Physics - Body

- AllowRotation: Allow this Body to be rotated? Default: true

sprite.body.allowRotation = false;

- AllowGravity: Allow this Body to be influenced by gravity? Default: true

sprite.body.allowGravity = false;

- CollideWorldBounds: Should the Body collide with the World bounds?

sprite.body.collideWorldBounds = true;

- Immovable: An immovable Body will not receive any impacts from other bodies.

sprite.body.immovable = true;

Physics – Arcade Physics - Body

```
//A local gravity applied to this Body.
sprite.body.gravity.set(0, 180);
sprite.body.gravity.x = 0;
sprite.body.gravity.y = 180;
//The drag applied to the motion of the Body.
sprite.body.drag.set(100);
//The acceleration is the rate of change of the velocity. Measured in pixels per second squared.
sprite.body.acceleration.set(0);
//The velocity, or rate of change in speed of the Body. Measured in pixels per second.
sprite.body.velocity.x = 0;
sprite.body.velocity.y = 0;
```

Physics – P2

P2.JS is a full-body physics system, with constraints, springs, polygon support and more.

If you need to model springs (eg. something swinging like a pendulum), constraints on forces, and arbitrary polygon shapes (eg. tetrahedron), this sounds like what you want. If you want a frame of reference, think of something like Angry Birds.

Physics – P2 - Instantiation

```
//Start the physics engine
game.physics.startSystem(Phaser.Physics.P2JS);

//attach P2 physics to object called “sprite”
game.physics.p2.enable(sprite);
```

Physics – P2 - Methods

// Turn on impact events for the world, without this we get no collision callbacks

game.physics.p2.setImpactEvents(true);

Create a collision group in P2

//1. CreateCollisionGroup(object) - Creates a new Collision Group and optionally applies it to the given object.

var playerCollisionGroup = game.physics.p2.createCollisionGroup();

//2. Adjust the bounds to use its own collision group. This code is essential when applying physics to a group

game.physics.p2.updateBoundsCollisionGroup();

//3. Tell the playerOne to use the playerCollisionGroup

playerOne.body.setCollisionGroup(playerCollisionGroup);

//4. Set trigger when the object collides: collides(group, callback, callbackContext, shape)

bullet.body.collides(playerCollisionGroup, hitPlayer, this);

Physics – P2 - Methods

- createDistanceConstraint(bodyA, bodyB, distance, localAnchorA, localAnchorB, maxForce): Creates a constraint that tries to keep the distance between two bodies constant. The parameters localAnchorA, localAnchorB and maxForce are optionals.

```
var constraint = game.physics.p2.createDistanceConstraint(sprite1,
sprite2, 150);
```

- createGearConstraint(bodyA, bodyB, angle, ratio): Creates a constraint that tries to keep the distance between two bodies constant the same rotational direction. The ratio and angle are optional

```
var constraint = game.physics.p2.createGearConstraint(sprite,
sonic1, 0, 1);
```

Physics – P2 - Methods

// createSpring(bodyA, bodyB, restLength, stiffness, damping, worldA, worldB, localA, localB): Creates a linear spring, connecting two bodies. A spring can have a resting length (default: 1), a stiffness (default: 100) and damping (default: 1). For parameters worldA, worldB, localA and localB, it determines where the spring to be attached.

var mouseSpring = game.physics.p2.createSpring(mouseBody, characterBody, 0, 30, 1);

//Remove spring

game.physics.p2.removeSpring(mouseSpring);

Physics – P2 - Methods

- createLockConstraint(bodyA, bodyB, offset, angle, maxForce): Locks the relative position between two bodies.The offset, angle and maxForce are optional. Offset parameter represents the offset distance between of bodyB in bodyA's frame.

```
var constraint = game.physics.p2.createLockConstraint(sprite, vu1,
[0, 80], 0);
```

Physics – P2 – Material Methods

//1. createMaterial(name, body): Creates a Material. Materials are applied to Shapes owned by a Body and can be set with Body.setMaterial(). Materials are a way to control what happens when Shapes collide. Combine unique Materials together to create Contact Materials. Contact Materials have properties such as friction and restitution that allow for fine-grained collision control between different Materials. Parameter *name* represents a unique name and if parameter *body* is given, it will assign the newly created Material to the Body shapes.

```
   var spriteMaterial =
game.physics.p2.createMaterial('spriteMaterial', player.body);

    var boxMaterial =
game.physics.p2.createMaterial('worldMaterial');
```

Physics – P2 – Material Methods

//2. createContactMaterial(materialA, materialB, options): Creates a Contact Material from the two given Materials. You can then edit the properties of the Contact Material directly.

```
var groundPlayerCM =
game.physics.p2.createContactMaterial(spriteMaterial,
worldMaterial, { friction: 0.0 });
```

Physics – P2 – Material Methods

//3. Set the options

Creates a Contact Material from the two given Materials. You can then edit the properties of the Contact Material directly.

groundPlayerCM.friction = 0.3; // Friction to use in the contact of these two materials.

groundPlayerCM.restitution = 1.0; // Restitution (i.e. how bouncy it is!) to use in the contact of these two materials.

groundPlayerCM.stiffness = 1e7; // Stiffness of the resulting ContactEquation that this ContactMaterial generate.

groundPlayerCM.relaxation = 3; // Relaxation of the resulting ContactEquation that this ContactMaterial generate.

groundPlayerCM.frictionStiffness = 1e7; // Stiffness of the resulting FrictionEquation that this ContactMaterial generate.

groundPlayerCM.frictionRelaxation = 3; // Relaxation of the resulting FrictionEquation that this ContactMaterial generate.

groundPlayerCM.surfaceVelocity = 0; // Will add surface velocity to this material. If bodyA rests on top if bodyB, and the surface velocity is positive, bodyA will slide to the right.

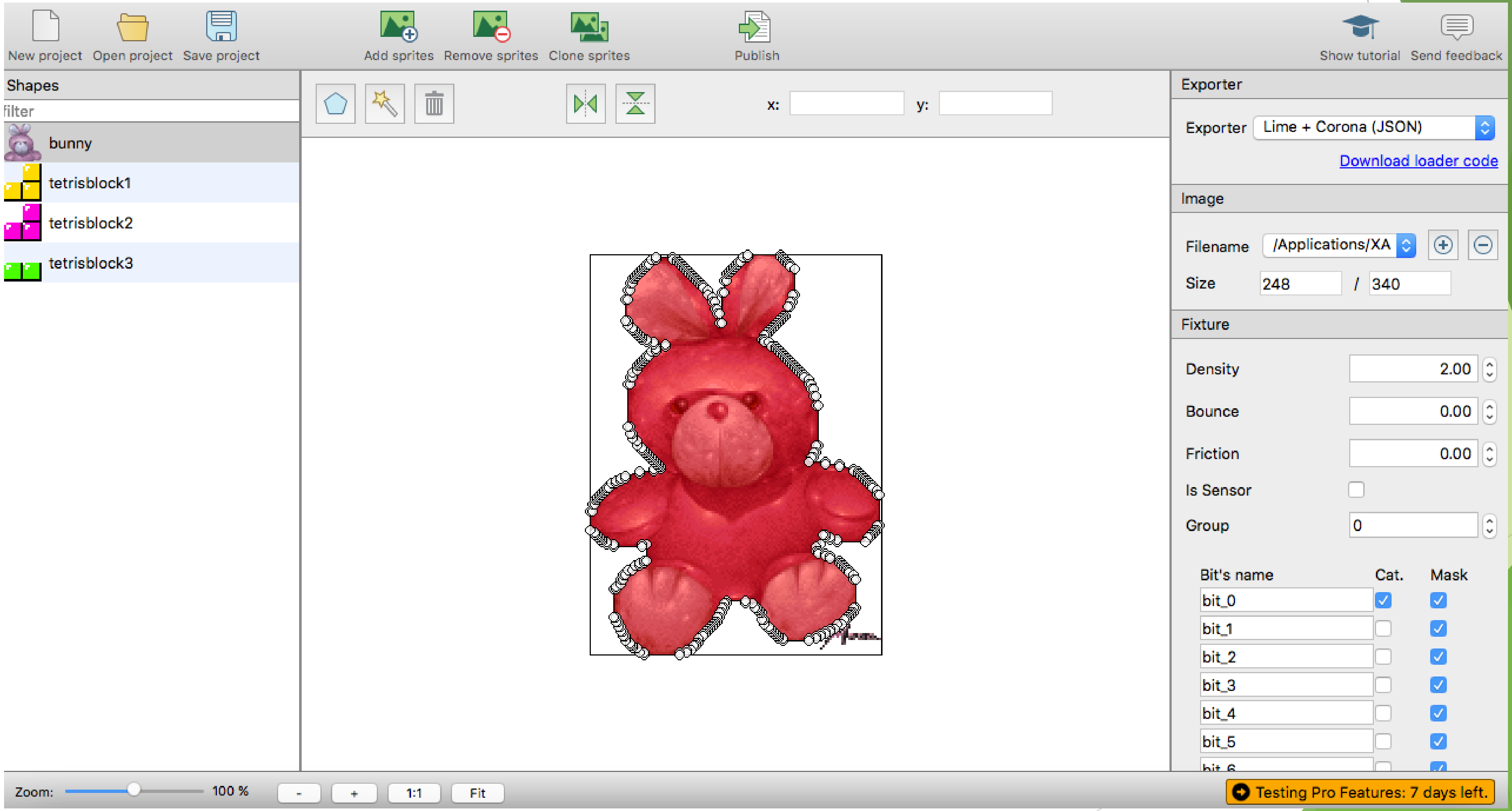
Physics – P2 – Methods - Body

```
ship.body.rotateLeft(100);
ship.body.rotateRight(100);
ship.body.thrust(400);
ship.body.reverse(400);
sprite.body.moveLeft(400);
sprite.body.moveRight(400);
sprite.body.moveUp(400);
sprite.body.moveDown(400);
```

Physics – P2 – Methods - Body

```
obj1.body.force.x
obj1.body.force.y
ship.body.collideWorldBounds = true;
sprite.body.velocity.x = 200;
sprite.body.velocity.y = -200;
ship.body.setZeroVelocity();
sprite.body.setZeroDamping();
sprite.body.fixedRotation = true;
```

Physics – P2 – Call JSON Data



Exporter

Exporter: Lime + Corona (JSON)

Download loader code

Image

Filename: /Applications/XA

Size: 248 / 340

Fixture

Density: 2.00

Bounce: 0.00

Friction: 0.00

Is Sensor: ☐

Group: 0

Bit's name	Cat.	Mask
bit_0	✓	✓
bit_1	☐	✓
bit_2	☐	✓
bit_3	☐	✓
bit_4	☐	✓
bit_5	☐	✓
bit_6	☐	✓

Physics – P2 – Call JSON Data

1. Add the sprites to the Physics Editor

2. Click on the Shape Tracer (the “wand” icon) which will automatically map the edges of the sprite and also apply physics on it

3. Publish the JSON containing the physics data

4. Load resource

```
game.load.physics('physicsData', 'assets/physics/sprites.json');
```

5. Apply the physics

```
tetris1.body.clearShapes();
tetris1.body.loadPolygon('physicsData', 'tetrisblock1');
```

http://phaser.io/examples/v2/p2-physics/load-polygon-1

Physics – P2 – Tilemap

http://phaser.io/examples/v2/p2-physics/tilemap

http://phaser.io/examples/v2/p2-physics/tilemap-gravity

Physics – Ninja Physics

Ninja Physics allows for complex tiles and slopes, perfect for level scenery

Ninja Physics will handle rotations (so it can do slopes and complex tiles). This is a more flexible (and probably more accurate) physics model;

It's probably slower.

Most of its functions are similar to Arcade physics

Physics – Ninja Physics

//Implementing Ninja physics on tiles against the character

//First way: Using AABB

1. Initate the physics system

```
game.physics.startSystem(Phaser.Physics.NINJA);
```

2. Enable ninja on the sprite and creates an AABB around it

```
game.physics.ninja.enableAABB(sprite1);
```

Physics – Ninja Physics

3. Enable tile

enableTile(object, id, children): This will create a Ninja Physics Tile body on the given game object. There are 34 different types of tile you can create, including 45 degree slopes, convex and concave circles and more. The id parameter controls which Tile type is created, but you can also change it at run-time. Note that for all degree based tile types they need to have an equal width and height. If the given object doesn't have equal width and height it will use the width. A game object can only have 1 physics body active at any one time, and it can't be changed until the object is destroyed.

Object: The game object to create the physics body on. Can also be an array or Group of objects, a body will be created on every child that has a body property.

Id (optional): The type of Tile this will use, i.e. Phaser.Physics.Ninja.Tile.SLOPE_45DEGpn, Phaser.Physics.Ninja.Tile.CONVEXpp, etc. Default: 1

Children (optional): Should a body be created on all children of this object? If true it will recurse down the display list as far as it can go.

```
game.physics.ninja.enableTile(tile);
```

Physics – Ninja Physics

//Second way: Apply physics to tile directly

1. Initate the physics system

```
game.physics.startSystem(Phaser.Physics.NINJA);
```

2. Load tiles resources

```
    game.load.tilemap('map', 'assets/tilemaps/maps/ninja-tilemap.json', null, Phaser.Tilemap.TILED_JSON);
    game.load.image('kenney', 'assets/tilemaps/tiles/kenney.png');
```

Physics – Ninja Physics

3. Convert Tilemap

convertTilemap(map, layer, slopeMap): Goes through all tiles in the given Tilemap and TilemapLayer and converts those set to collide into physics tiles. Only call this after you have specified all of the tiles you wish to collide with calls like Tilemap.setCollisionBetween, etc. Every time you call this method it will destroy any previously created bodies and remove them from the world. Therefore understand it's a very expensive operation and not to be done in a core game update loop.

In Ninja the Tiles have an ID from 0 to 33, where 0 is 'empty', 1 is a full tile, 2 is a 45-degree slope, etc. You can find the ID list either at the very bottom of Tile.js, or in a handy visual reference in the resources/Ninja Physics Debug Tiles folder in the repository. The slopeMap parameter is an array that controls how the indexes of the tiles in your tilemap data will map to the Ninja Tile IDs. For example if you had 6 tiles in your tileset: Imagine the first 4 should be converted into fully solid Tiles and the other 2 are 45-degree slopes. Your slopeMap array would look like this: [1, 1, 1, 1, 2, 3]. Where each element of the array is a tile in your tilemap and the resulting Ninja Tile it should create.

map: The Tilemap to get the map data from.

layer: The layer to operate on. If not given will default to map.currentLayer.

slopemap: The tilemap index to Tile ID map.

```
var slopeMap = { '32': 1, '77': 1, '95': 2, '36': 3, '137': 3, '140': 2 };

tiles = game.physics.ninja.convertTilemap(map, layer, slopeMap);
```

Physics – Ninja Physics

4. Create physics on the character

enableCircle(object, radius, children): This will create a Ninja Physics Circle body on the given game object. A game object can only have 1 physics body active at any one time, and it can't be changed until the object is destroyed.

Object: The game object to create the physics body on. Can also be an array or Group of objects, a body will be created on every child that has a body property.

Radius: The radius of the Circle.

Children (optional): Should a body be created on all children of this object? If true it will recurse down the display list as far as it can go. Default: true

```
game.physics.ninja.enableCircle(sprite1, sprite1.width / 2);
```

5. Check if the circle collide with the tile

collideCircleVsTile(t): Collides this Circle with a Tile. Parameter *t* is the tile involved in the collision.

```
//check when the circled character collide with all tiles

for (var i = 0; i < game.collideLayer.length; i++){

game.player.body.collideCircleVsTile(game.collideLayer[i].tile);

}
```

Resources

http://phaser.io/examples/v2/category/arcade-physics

http://phaser.io/examples/v2/category/p2-physics

http://phaser.io/examples/v2/category/ninja-physics

http://phaser.io/docs/2.4.4/index

http://www.codevinsky.com/phaser-2-0-tutorial-flappy-bird-part-2/

http://gamedev.stackexchange.com/questions/72930/difference-between-arcade-p2-and-ninja-physics-in-phaser

http://www.html5gamedevs.com/topic/4518-explaining-phaser-2s-multiple-physics-systems/

http://www.joshmorony.com/building-a-running-platformer-in-phaser-with-ninja-physics/

Particles

//1. adding particle to the game: **add.emitter(x, y, maxParticles)**

var emitter = game.add.emitter(game.world.centerX, game.world.centerY, 250);

//2. make particles: **makeParticles(keys, frames, quantity, collide, collideWorldBounds)**

Keys: A string or an array of strings that the particle sprites will use as their texture. If an array one is picked at random.

Frames (optional): A frame number, or array of frames that the sprite will use. If an array one is picked at random. Default: 0

Quantity (optional): The number of particles to generate. If not given it will use the value of Emitter.maxParticles. If the value is greater than Emitter.maxParticles it will use Emitter.maxParticles as the quantity.

Collide (optional): If you want the particles to be able to collide with other Arcade Physics bodies then set this to true. Default: false

CollideWorldBounds (optional): A particle can be set to collide against the World bounds automatically and rebound back into the World if this is set to true. Otherwise it will leave the World. Default: false

emitter.makeParticles('balls', [0, 1, 2, 3, 4, 5]);

Particles

- // 3. Start particle. There are 2 ways to start: by using start method and by using flow method

start(explode, lifespan, frequency, quantity, forceQuantity)
Call this function to start emitting particles.

Explode(optional): Whether the particles should all burst out at once (true) or at the frequency given (false). Default: true
Lifespan(optional): How long each particle lives once emitted in ms. 0 = forever. Default: 0
Frequency(optional): Ignored if Explode is set to true. Frequency is how often to emit 1 particle. Value given in ms. Default: 250
Quantity(optional): How many particles to launch. 0 = "all of the particles" which will keep emitting until Emitter.maxParticles is reached. Default: 0
ForceQuantity(optional): If true and creating a particle flow, the quantity emitted will be forced to the be quantity given in this call. This can never exceed Emitter.maxParticles. Default: true

```
emitter.start(false, 5000, 20);
```

Particles

flow(lifespan, frequency, quantity, total, immediate)
Call this function to start emitting a flow of particles at the given frequency. It will carry on going until the total given is reached. Each time the flow is run the quantity number of particles will be emitted together. If you set the total to be 20 and quantity to be 5 then flow will emit 4 times in total (4 x 5 = 20 total) If you set the total to be -1 then no quantity cap is used and it will keep emitting.

Lifespan (optional): How long each particle lives once emitted in ms. 0 = forever. Default: 0
Frequency (optional): Frequency is how often to emit the particles, given in ms. Default: 250
Quantity (optional): How many particles to launch each time the frequency is met. Can never be > Emitter.maxParticles. Default: 1
Total (optional): How many particles to launch in total. If -1 it will carry on indefinitely. Default: -1
Immediate (optional): Should the flow start immediately (true) or wait until the first frequency event? (false). Default: true

```
emitter.flow(2000, 500, 5, -1);
```

Particles - Methods

- emitter.minParticleSpeed.setTo(-400, -400); //The minimum possible velocity of a particle.
- emitter.maxParticleSpeed.setTo(400, 400); //The maximum possible velocity of a particle.
- emitter.minParticleScale = 0.1; // The minimum possible scale of a particle. This is applied to the X and Y axis. If you need to control each axis see minParticleScaleX. Default: 1
- emitter.maxParticleScale = 0.5; // The maximum possible scale of a particle. This is applied to the X and Y axis. If you need to control each axis see maxParticleScaleX. Default: 1

Particles - Methods

- emitter.setRotation(0, 0); //A more compact way of setting the angular velocity constraints of the particles. Parameters: *min* and *max*
- emitter.gravity = -200; // Sets the body.gravity.y of each particle sprite to this value on launch. Default: 100
- emitter.setYSpeed(20, 100); // A more compact way of setting the Y velocity range of the emitter. Parameters: *min* and *max*
- emitter.setXSpeed(120, 10); // A more compact way of setting the X velocity range of the emitter. Parameters: *min* and *max*

Particles - Methods

- emitter.width = game.world.width * 1.5; // Gets or sets the width of the Emitter. This is the region in which a particle can be emitted.
- emitter.height = game.world.height * 1.5; // Gets or sets the height of the Emitter. This is the region in which a particle can be emitted.
- emitter.minRotation = 0; // The minimum possible angular velocity of a particle.
- emitter.maxRotation = 40; // The maximum possible angular velocity of a particle.
- emitter.setRotation(0, 0); //A more compact way of setting the angular velocity constraints of the particles. Parameters: *min* and *max*

Particles - Methods

- **setAlpha(min, max, rate, ease, yoyo)**

  //A more compact way of setting the alpha constraints of the particles. The rate parameter, if set to a value above zero, lets you set the speed at which the Particle change in alpha from min to max. If rate is zero, which is the default, the particle won't change alpha - instead it will pick a random alpha between min and max on emit.

Min(optional): The minimum value for this range. Default: 1

Max(optional): The maximum value for this range. Default: 1

Rate(optional): The rate (in ms) at which the particles will change in alpha from min to max, or set to zero to pick a random alpha between the two. Default: 0

Ease(optional): If you've set a rate > 0 this is the easing formula applied between the min and max values. Default: Phaser.Easing.Linear.None

Yoyo(optional): If you've set a rate > 0 you can set if the ease will yoyo or not (i.e. ease back to its original values). Default: false

```
emitter.setAlpha(0.3, 0.8);
```

Particles - Methods

- **setScale(minX, maxX, minY, maxY, rate, ease, yoyo)**

  //A more compact way of setting the scale constraints of the particles. The rate parameter, if set to a value above zero, lets you set the speed and ease which the Particle uses to change in scale from min to max across both axis. If rate is zero, which is the default, the particle won't change scale during update, instead it will pick a random scale between min and max on emit.

MinX(optional): The minimum value of Particle.scale.x.. Default: 1

MaxX(optional): The maximum value of Particle.scale.x. Default: 1

MinY(optional): The minimum value of Particle.scale.y. Default: 1

MaxY(optional): The maximum value of Particle.scale.y.. Default: 1

Rate(optional): The rate (in ms) at which the particles will change in scale from min to max, or set to zero to pick a random size between the two. Default: 0

Ease(optional): If you've set a rate > 0 this is the easing formula applied between the min and max values. Default: Phaser.Easing.Linear.None

Yoyo(optional): If you've set a rate > 0 you can set if the ease will yoyo or not (i.e. ease back to its original values). Default: false

```
emitter.setScale(0.5, 0.5, 1, 1);
```

Filters

- Filters in Phaser use GLSL
- GLSL is an high level openGL Shading Language which is based on the syntax of the C programming language
- Some GLSL resources:
  - http://glslsandbox.com
  - https://www.shadertoy.com/
  - https://cdn.rawgit.com/photonstorm/phaser/master/filters/Tunnel.js

Filters

```
//1. Load the GLSL
 game.load.shader(‘shaderVarName', 'assets/shaders/bacteria.frag');
//2. Instantiate filter
filterVar = new Phaser.Filter(game, null,
    game.cache.getShader(shaderVarName '));
//3. Add filter
filterVar.addToWorld(0, 0, 800, 600);
//4. Assign the filter to a sprite
sprite.filters = [filterVar];
//5. Run the filter
filterVar.update();
```

Resources

- http://phaser.io/examples/v2/category/filters
- http://phaser.io/examples/v2/category/particles

- http://phaser.io/docs/2.4.4/Phaser.Particles.html
- http://phaser.io/docs/2.4.4/Phaser.Particles.Arcade.Emitter.html
- http://phaser.io/docs/2.4.4/Phaser.Particle.html