Creating Particles with Particle Systems

# Introduction

Particles are textured images displayed at positions with no defined dimensions that often participate in the physics simulation. The textured images facilitate representation of interesting effects while positions without dimensions allow simple and efficient physics simulations. A particle system controls and strategizes the generation of particles with specific appearance and behavior. A particle system provides a game engine with the ability to implement dynamic-looking reactions. This includes common effects such as explosions, fire and effects associated with casting spells. Building upon the physics component, you will create a particle system that can support diverse gameplay requirements.

In this chapter, you will create a simple and flexible particle system that includes the basic functionality required to achieve common effects, such as explosions and spell effects. Additionally, you will implement a particle shader to properly blend your particles within your scenes. The particles will collide and interact accordingly with the RigidShape objects. You will also discover and define the need for particles that are generated a specific point over a period of time such as a campfire or torch these are often called particle emitters.

# Particles and Particle Systems

A particle is a textured position with no defined dimensions. This description may seem contradictory because you have learned that a texture is an image and images are always defined by a width and height and will definitely occupy areas. The important clarification is that the game engine logic processes a particle as a position with no area, and the drawing system displays the particle as a texture. In this way, even though an actual displayed area is shown, the width and height dimensions of the texture are ignored by the underlying logic.

**Goal is to show the fundamental, covering one specific type of particle focusing on how to create, how to iterate through, and what effect can look like. free to examine experiment other types of particle/**

In addition to a position, a particle also has properties such as life span, color (for tinting the texture), and size (for scaling the texture). A particle system is a collection of particles with the same set of properties. By strategically spawning, manipulating the properties of, and removing particles in a particle system, interesting visual effects such as fire or explosions can be simulated.

**Define what is a particle system**

In the following project, you will first learn about the support for drawing a particle object. After that, you will examine the details of how to create an actual particle object and define its behaviors. A particle is a new type of object for your game engine and requires the support of the entire drawing system of your engine, including custom GLSL shaders, default sharable shader instance, and a new Shader/Renderable pair.

## The Particles Project

This project demonstrates how to implement a particle system to simulate explosion or spell-like effects. You can see an example of this project running in Figure 10-1. The source code of this project is located in the chapter10/10.1.particles folder.



Figure 10-1. Running the Particles project

The controls of the project are as follows:

* WASD keys: Moves the Hero object
* Q key: Spawns particles at the current mouse position

The goals of the project are as follows:

* To understand the details of how to draw a particle and define its behaviors
* To experience implementing a particle system

You can find the following external resources in the assets folder: the fonts folder that contains the default system fonts, the particles folder that contains the particle.png the default particle texture and the four texture images (minion\_sprite.png, which defines the sprite elements for the hero and the minions; platform.png, which defines the platforms and floor and ceiling tiles; wall.png, which defines the walls; and the target.png).

### Creating GLSL Particle Fragment Shader

Particles are textured positions with no areas. However, as discussed, your engine will draw each particle as a textured rectangle. For this reason, you can simply reuse the existing texture vertex shader texture\_vs.glsl. When it comes to the actual computation of each pixel color, a new fragment shader, particle\_fs.glsl, must be created to properly simulate the intense brightness of fire and explosions. The new fragment shader aggressively composites colors to create oversaturation, where the resulting values of the RGB components quickly become the maximum displayable value of 1.0.

1. Under the src/glsl\_shaders folder, add a new file and name it particle\_fs.glsl.
2. Similar to the texture fragment shader defined in texture\_fs.glsl, you need to declare uPixelColor and vTexCoord to receive these values from the game engine and define the uSampler to sample the texture.

precision mediump float;

// sets the precision for floating point computation

// The object that fetches data from texture.

// Must be set outside the shader.

uniform sampler2D uSampler;

// Color of pixel

uniform vec4 uPixelColor;

// The "varying" keyword is for signifying that the texture coordinate will be

// interpolated and thus varies.

varying vec2 vTexCoord;

1. Now implement the main function to accumulate colors aggressively through multiplication.

void main(void) {

// texel color look up based on interpolated UV value in vTexCoord

vec4 c = texture2D(uSampler, vec2(vTexCoord.s, vTexCoord.t));

vec3 r = vec3(c) \* c.a \* vec3(uPixelColor);

vec4 result = vec4(r, uPixelColor.a);

gl\_FragColor = result;

}

### Defining a Default ParticleShader Instance

You can now modify the engine to support the initializing, loading, and unloading of a new ParticleShader.

1. Begin by adding a variable for the particle shader in the shader\_resources.js file located in the src/engine/core folder. Also, define an accessor function as shown here:

// Particle Shader

let kParticleFS = "src/glsl\_shaders/particle\_fs.glsl";

let mParticleShader = null;

function getParticleShader() { return mParticleShader }

1. Now instantiate a new particle shader in the createShaders() function, as shown here:

function createShaders() {

mConstColorShader = new SimpleShader(kSimpleVS, kSimpleFS);

mTextureShader = new TextureShader(kTextureVS, kTextureFS);

mSpriteShader = new SpriteShader(kTextureVS, kTextureFS);

mLineShader = new LineShader(kSimpleVS, kLineFS);

mLightShader = new LightShader(kTextureVS, kLightFS);

mIllumShader = new IllumShader(kTextureVS, kIllumFS);

mShadowCasterShader = new ShadowCasterShader(kTextureVS, kShadowCasterFS);

mShadowReceiverShader = new SpriteShader(kTextureVS, kShadowReceiverFS);

mParticleShader = new TextureShader(kTextureVS, kParticleFS);

}

1. In the init() function, add the following code to properly load the file:

function init() {

let loadPromise = new Promise(

async function(resolve) {

await Promise.all([

text.load(kSimpleFS),

text.load(kSimpleVS),

text.load(kTextureFS),

text.load(kTextureVS),

text.load(kLineFS),

text.load(kLightFS),

text.load(kIllumFS),

text.load(kShadowCasterFS),

text.load(kShadowReceiverFS),

text.load(kParticleFS)

]);

resolve();

}).then(

function resolve() { createShaders(); }

);

map.pushPromise(loadPromise);

}

1. In the cleanUp() function, add the following line of code to unload the file when it is no longer needed:

function cleanUp() {

mConstColorShader.cleanUp();

mTextureShader.cleanUp();

mSpriteShader.cleanUp();

mLineShader.cleanUp();

mLightShader.cleanUp();

mIllumShader.cleanUp();

mShadowReceiverShader.cleanUp();

mShadowCasterShader.cleanUp();

mParticleShader.cleanUp();

text.unload(kSimpleVS);

text.unload(kSimpleFS);

text.unload(kTextureVS);

text.unload(kTextureFS);

text.unload(kLineFS);

text.unload(kLightFS);

text.unload(kIllumFS);

text.unload(kShadowCasterFS);

text.unload(kShadowReceiverFS);

text.unload(kParticleFS);

}

1. Don’t forget to add the export.

export {init, cleanUp,

getConstColorShader, getTextureShader, getSpriteShader, getLineShader,

getLightShader, getIllumShader, getShadowReceiverShader, getShadowCasterShader,

getParticleShader}

### Creating the ParticleRenderable Object

Recall that a Shader/Renderable pair of objects must be defined to interface the GLSL shader to the game engine. With the default ParticleShader object defined to interface to the GLSL particle\_fs shader, you can now create a new Renderable object to support the drawing of particles. Fortunately, the detailed behaviors of a particle, or a textured position, is identical to that of a TextureRenderable with the exception of the different shader. As such, the definition of the ParticleRenderable object is trivial.

1. Under the src/engine/renderables folder, add a new file and name it particle\_renderable.js.
2. Define the new renderable object as a simple subclass of the TextureRenderable, with the exception of a different default shader.

import \* as defaultShaders from "../core/shader\_resources.js";

import TextureRenderable from "./texture\_renderable.js";

class ParticleRenderable extends TextureRenderable {

constructor(myTexture) {

super(myTexture);

this.\_setShader(defaultShaders.getParticleShader());

}

}

export default ParticleRenderable;

### Defining the Particle and Particle Game Object

With the infrastructure for drawing a particle object defined, it is time to create the actual particle and define its behaviors.

#### Creating a Particle

To support moving particles around the scene, their movements can be approximated with the Symplectic Euler Integration.

1. Begin by creating a new subfolder called particles under the src/engine folder. In this folder, create a new file called particle.js.
2. Create a new class called Particle and add the constructor and variables for position, velocity, acceleration, drag, and drawing variables for marking its position (for debugging).

import \* as loop from "../core/loop.js";

import \* as particleSystem from "../components/particle\_system.js";

import ParticleRenderable from "../renderables/particle\_renderable.js";

import \* as debugDraw from "../core/debug\_draw.js";

let kSizeFactor = 0.2;

class Particle {

constructor(texture, x, y, life) {

this.mRenderComponent = new ParticleRenderable(texture);

this.setPosition(x, y);

// position control

this.mVelocity = vec2.fromValues(0, 0);

this.mAcceleration = particleSystem.getSystemAcceleration();

this.mDrag = 0.95;

// Color control

this.mDeltaColor = [0, 0, 0, 0];

// Size control

this.mSizeDelta = 0;

// Life control

this.mCyclesToLive = life;

}

…

}

export default Particle;

1. Create a drawMarker() function to draw the position of the particle with a simple X as well as a simple draw() function. This function is to support the debugging of a particle system.

drawMarker(aCamera) {

let size = this.getSize();

debugDraw.drawCrossMarker(aCamera, this.getPosition(), size[0] \* kSizeFactor, [0, 1, 0, 1]);

}

draw(aCamera) {

this.mRenderComponent.draw(aCamera);

}

1. You can now implement the update() function for calculating the particle’s position. This is a straightforward implementation of the Symplectic Euler Integration. The mDrag variable simulates drags on the particles.

update() {

this.mCyclesToLive--;

let dt = loop.getUpdateIntervalInSeconds();

// Symplectic Euler

// v += a \* dt

// x += v \* dt

let p = this.getPosition();

vec2.scaleAndAdd(this.mVelocity, this.mVelocity, this.mAcceleration, dt);

vec2.scale(this.mVelocity, this.mVelocity, this.mDrag);

vec2.scaleAndAdd(p, p, this.mVelocity, dt);

// update color

let c = this.mRenderComponent.getColor();

vec4.add(c, c, this.mDeltaColor);

// update size

let xf = this.mRenderComponent.getXform();

let s = xf.getWidth() \* this.mSizeDelta;

xf.setSize(s, s);

}

1. Define simple get and set accessors. These functions are straightforward and are not listed here.

#### Creating the ParticleGameObject

With the actual particle object defined, you can now create the ParticleGameObject class to abstract the particle’s behavior and to support the drawing of the particle.

1. Under the src/Engine/Particles folder, add a new file and name it ParticleGameObject.js. Remember to load this new source file in index.html.
2. Define a constructor to initialize variables for manipulating the color, size, and life span of the particle.
3. Create a function for setting the final color of the particle. This function computes the required rate of change such that the color of the particle will be constantly changing during each update cycle unit the particle expires.
4. Define accessor functions to set the particle size change rate and to test whether the particle has expired. An expired particle should be removed from the system.
5. Add an update() function to change the color and size of the particle.

#### Creating the ParticleSet

To work with a collection of particles, you can now create the ParticleSet to support conveniently looping over sets of Particles.

1. Under the src/engine/particles folder, add a new file and name it particle\_set.js.
2. Define ParticleSet to be a subclass of GameObjectSet.

import \* as glSys from "../core/gl.js";

import GameObjectSet from "../game\_objects/game\_object\_set.js";

class ParticleSet extends GameObjectSet {

constructor() {

super();

}

…

}

export default ParticleSet;

1. Override the draw() function of GameObjectSet to ensure particles are drawn with additive blending. Recall that the default gl.blendFunc() setting utilizes the alpha channel value to implement transparency. This is referred to as alpha blending. In this case, the gl.blendFunc() setting results in a simple accumulation of colors without considering the alpha channel. This is referred to as additive blending. As mentioned, to properly simulate the intense brightness of fire and explosions, it is desirable to aggressively accumulate colors such that the RGB components can quickly reach the maximum displayable value of 1.0. Additive blending facilitates the aggressive accumulation.

draw(aCamera) {

let gl = glSys.get();

gl.blendFunc(gl.ONE, gl.ONE); // for additive blending!

super.draw(aCamera);

gl.blendFunc(gl.SRC\_ALPHA, gl.ONE\_MINUS\_SRC\_ALPHA); // restore alpha blending

}

drawMarkers(aCamera) {

let i;

for (i = 0; i < this.mSet.length; i++) {

this.mSet[i].drawMarker(aCamera);

}

}

1. Override the update() function to ensure expired particles are removed.

update() {

super.update();

// Cleanup Particles

let i, obj;

for (i = 0; i < this.size(); i++) {

obj = this.getObjectAt(i);

if (obj.hasExpired()) {

this.removeFromSet(obj);

}

}

}

### Defining the Engine Particle Component

With the drawing and behavior infrastructures defined, you can now define the engine component to manage the particle system.

1. In the src/engine/components folder, add a new file and name it particle\_system.js.
2. Define a variable for system acceleration.

let mSystemAcceleration = [30, -50.0];

1. Finally, implement the necessary get and set accessors for the system’s acceleration and remember to export the necessary functions.

function getSystemAcceleration() { return vec2.clone(mSystemAcceleration); }

function setSystemAcceleration(x, y) {

mSystemAcceleration[0] = x;

mSystemAcceleration[1] = y;

}

export {getSystemAcceleration, setSystemAcceleration}

### index.js and default resource changes

### Testing the Particle System

The test should verify two main goals. First, the implemented particle system is capable of generating visually pleasant effects. Second, the particles are handled correctly, as in created and destroyed properly and produce the expected behaviors. The test case is based mainly on the previous project with one simple modification—invoking the \_createParticle() function when the Q key is pressed. Please consult the source code for the details of the implementation.

The \_createParticle() function listed in the following configures and creates particles. There are two important observations. First, the random() function is used many times to configure each created Particle. Particle systems utilize large numbers of similar yet slightly different particles to build and convey a sense of the desired visual effect. It is important to avoid patterns of any sort, and this use of randomness is an important rule to follow. Second, there are many seemingly arbitrary numbers used in the configuration, such as setting the life of the particle to be between 30 and 230 or setting the final red component to a number between 3.5 and 4.5. This, unfortunately is the nature of working with particle systems: there is quite a bit of ad hoc trying. Commercial game engines typically alleviate this difficulty by releasing a collection of preset values for their particle systems. In this way, game designers can fine-tune specific desired effects by tweaking the provided presets.

function \_createParticle(atX, atY) {

let life = 30 + Math.random() \* 200;

let p = new engine.Particle(engine.defaultResources.getDefaultPSTexture(), atX, atY, life);

p.setColor([1, 0, 0, 1]);

// size of the particle

let r = 5.5 + Math.random() \* 0.5;

p.setSize(r, r);

// final color

let fr = 3.5 + Math.random();

let fg = 0.4 + 0.1 \* Math.random();

let fb = 0.3 + 0.1 \* Math.random();

p.setFinalColor([fr, fg, fb, 0.6]);

// velocity on the particle

let fx = 10 - 20 \* Math.random();

let fy = 10 \* Math.random();

p.setVelocity(fx, fy);

// size delta

p.setSizeDelta(0.98);

return p;

}

### Observations

Run the project and press the Q key to observe the generated particles. It appears as though there is combustion occurring underneath the mouse pointer. Hold the Q key and move the mouse pointer around slowly to observe the combustion following the mouse as though there is an engine generating flames under the mouse pointer. If you move the mouse pointer quickly, you will observe individual pink circles changing color while dropping toward the floor. Particle systems must be fine-tuned for each individual situation for example if you were to attempt a fire-like system you would need to make the particles rise, change the color and use a flame like texture.

**Engine can define subclasses of Particle, with customized behavior, Shader always glows does not need to be so.**

**\_CreateParticle() function can have variety**

# Particle Collision

In this section, you will create a simple particle system that will interact by colliding with the RigidShape objects.

## The Particles Collisions Project

This project demonstrates how to implement a particle collision system that interacts with your existing implementation of RigidShape. You can see an example of this project running in Figure 10-2. The source code of this project is located in the chapter10/10.2.particles\_collisions folder.



Figure 10-2. Running the Particles Collisions project

The controls of the project are as follows:

* Q key: Spawns particles at the current mouse position

The goals of the project are as follows:

* To understand the details of how to add functionality between two existing engine systems (particles and physics)
* To build a particle engine component that supports interaction with RigidShape

You can find the following external resources in the assets folder: the fonts folder that contains the default system fonts, the particles folder that contains the particle.png the default particle texture and the four texture images (minion\_sprite.png, which defines the sprite elements for the hero and the minions; platform.png, which defines the platforms and floor and ceiling tiles; wall.png, which defines the walls; and the target.png).

### Modifying the Particle System

TEMP TEXT

1. TEMP TEXT

import Transform from "../utils/transform.js";

import RigidCircle from "../rigid\_shapes/rigid\_circle.js";

import CollisionInfo from "../rigid\_shapes/collision\_info.js";

let mXform = null; // for collision with rigid shapes

let mCircleCollider = null;

let mCollisionInfo = null;

let mFrom1to2 = [0, 0];

function init() {

mXform = new Transform();

mCircleCollider = new RigidCircle(mXform, 1.0);

mCollisionInfo = new CollisionInfo();

}

function resolveCirclePos(circShape, particle) {

let collision = false;

let pos = particle.getPosition();

let cPos = circShape.getCenter();

vec2.subtract(mFrom1to2, pos, cPos);

let dist = vec2.length(mFrom1to2);

if (dist < circShape.getRadius()) {

vec2.scale(mFrom1to2, mFrom1to2, 1/dist);

vec2.scaleAndAdd(pos, cPos, mFrom1to2, circShape.getRadius());

collision = true;

}

return collision;

}

function resolveRectPos(rectShape, particle) {

let collision = false;

let s = particle.getSize();

let p = particle.getPosition();

mXform.setSize(s[0], s[1]); // referred by mCircleCollision

mXform.setPosition(p[0], p[1]);

if (mCircleCollider.boundTest(rectShape)) {

if (rectShape.collisionTest(mCircleCollider, mCollisionInfo)) {

// make sure info is always from rect towards particle

vec2.subtract(mFrom1to2, mCircleCollider.getCenter(), rectShape.getCenter());

if (vec2.dot(mFrom1to2, mCollisionInfo.getNormal()) < 0)

mCircleCollider.adjustPositionBy(mCollisionInfo.getNormal(), -mCollisionInfo.getDepth());

else

mCircleCollider.adjustPositionBy(mCollisionInfo.getNormal(), mCollisionInfo.getDepth());

p = mXform.getPosition();

particle.setPosition(p[0], p[1]);

collision = true;

}

}

return collision;

}

// obj: a GameObject (with potential mRigidBody)

// pSet: set of particles (ParticleSet)

function resolveRigidShapeCollision(obj, pSet) {

let i, j;

let collision = false;

let rigidShape = obj.getRigidBody();

for (j = 0; j < pSet.size(); j++) {

if (rigidShape.getType() == "RigidRectangle")

collision = resolveRectPos(rigidShape, pSet.getObjectAt(j));

else if (rigidShape.getType() == "RigidCircle")

collision = resolveCirclePos(rigidShape, pSet.getObjectAt(j));

}

return collision;

}

// objSet: set of GameObjects (with potential mRigidBody)

// pSet: set of particles (ParticleSet)

function resolveRigidShapeSetCollision(objSet, pSet) {

let i, j;

let collision = false;

if ((objSet.size === 0) || (pSet.size === 0))

return false;

for (i=0; i<objSet.size(); i++) {

let rigidShape = objSet.getObjectAt(i).getRigidBody();

for (j = 0; j<pSet.size(); j++) {

if (rigidShape.getType() == "RigidRectangle")

collision = resolveRectPos(rigidShape, pSet.getObjectAt(j)) || collision;

else if (rigidShape.getType() == "RigidCircle")

collision = resolveCirclePos(rigidShape, pSet.getObjectAt(j)) || collision;

}

}

return collision;

}

export {init,

getSystemAcceleration, setSystemAcceleration,

resolveRigidShapeCollision, resolveRigidShapeSetCollision}

### Modifying the Loop

TEMP TEXT

1. TEMP TEXT

import \* as particleSystem from "../components/particle\_system.js";

1. TEMP TEXT

async function start(scene) {

if (mLoopRunning) {

throw new Error("loop already running")

}

mCurrentScene = scene;

mCurrentScene.load();

// Wait for any async requests before game-load

await map.waitOnPromises();

// System debug-drawing support

debugDraw.init();

// Other system init that can only occur after all resources are loaded

particleSystem.init();

// Now, initialize current scenes

mCurrentScene.init();

mPrevTime = performance.now();

mLagTime = 0.0;

mLoopRunning = true;

mFrameID = requestAnimationFrame(loopOnce);

}

### Testing the Particle System

TEMP TEXT

### Observations

TEMP TEXT

# Particle Emitters

With your current particle system implementation, you can create particles at a specific point and time. These particles when created can move and change based on their properties. However, particles can be created only when there is an explicit state change such as a key click. This becomes restricting when it is desirable to persist the generation of particles after the state change, such as an explosion that persists for a short while after the dye pack collides with the hero. A particle emitter addresses this issue by defining the functionality of generating particles over a time period.

## The Particle Emitters Project

This project demonstrates how to implement a particle emitter for your particle system to support emitting particles over time. You can see an example of this project running in Figure 10-3. The source code of this project is located in the chapter10/10.3.particle\_emitters folder.



Figure 9-11. Running the Particle Emitters project

The controls of the project are as follows:

* Q key: Spawns particles at the current mouse position

The goals of the project are as follows:

* To understand the need for particle emitters
* To experience implementing particle emitters

You can find the following external resources in the assets folder: the fonts folder that contains the default system fonts, the particles folder that contains the particle.png the default particle texture and the four texture images (minion\_sprite.png, which defines the sprite elements for the hero and the minions; platform.png, which defines the platforms and floor and ceiling tiles; wall.png, which defines the walls; and the target.png).

### Creating the ParticleEmitter Object

Recall that when working with particles, in general, it is important to avoid patterns. In this case, as the ParticleEmitter object generates new particles over time, it is often important to inject randomness to avoid a patterned look.

1. Under the src/engine/particles folder, add a new file and name it particle\_emitter.js.
2. Create a new class named ParticleEmitter and add a constructor to define and set the default number of particles to emit at each cycle, the emitter’s position, the number of particles left to be emitted, and mParticleCreator, the callback function for actual particles creation.

let kMinToEmit = 5; // Smallest number of particle emitted per cycle

class ParticleEmitter {

constructor(px, py, num, createrFunc) {

// Emitter position

this.mEmitPosition = [px, py];

// Number of particles left to be emitted

this.mNumRemains = num;

// Function to create particles (user defined)

this.mParticleCreator = createrFunc;

}

…

}

export default ParticleEmitter;

1. Define a function to return the functioning status for the emitter. When there are no more particles to emit, the emitters should be removed.

expired() { return (this.mNumRemains <= 0); }

1. Create a function to actually emit particles. Take note of the randomness in the number of particles that are actually emitted and the invocation of the mParticleCreator() callback function. With this design, there would be no patterns in the number of particles that are created over time. In addition, the emitter defines only the mechanisms of how, when, and where particles will be emitted and does not define the characteristics of the created particles. The function pointed to by mParticleCreator is responsible for defining that.

emitParticles(pSet) {

let numToEmit = 0;

if (this.mNumRemains < this.kMinToEmit) {

// If only a few are left, emits all of them

numToEmit = this.mNumRemains;

} else {

// Otherwise, emits about 20% of what's left

numToEmit = Math.trunc(Math.random() \* 0.2 \* this.mNumRemains);

}

// Left for future emitting.

this.mNumRemains -= numToEmit;

let i, p;

for (i = 0; i < numToEmit; i++) {

p = this.mParticleCreator(this.mEmitPosition[0], this.mEmitPosition[1]);

pSet.addToSet(p);

}

}

### Modifying the Particle Set

The defined ParticleEmitter object needs to be integrated into ParticleSet to manage the emitted particles.

1. Edit the particle\_set.js file; in the constructor, define and initialize a particle emitter set.

constructor() {

super();

this.mEmitterSet = [];

}

1. Define a function for instantiating a new emitter. Take note of the func parameter; this is the callback function that is responsible for the actual creation of individual Particle objects.

addEmitterAt(x, y, n, func) {

let e = new ParticleEmitter(x, y, n, func);

this.mEmitterSet.push(e);

}

1. Modify the update function to loop through the emitter set to generate new particles. Expired emitters are removed.

update() {

super.update();

// Cleanup Particles

let i, obj;

for (i = 0; i < this.size(); i++) {

obj = this.getObjectAt(i);

if (obj.hasExpired()) {

this.removeFromSet(obj);

}

}

// Emit new particles

for (i = 0; i < this.mEmitterSet.length; i++) {

let e = this.mEmitterSet[i];

e.emitParticles(this);

if (e.expired()) { // delete the emitter when done

this.mEmitterSet.splice(i, 1);

}

}

}

### Index.js stuff

### Testing the Particle Emitter

This is a straightforward testing of the correct functioning of the ParticleEmitter object. The two interesting implementation details are as follows:

In the Hero.js file, at the end of the update() function, when a collision between a dye pack and the hero is detected, a new ParticleEmitter object is instantiated with the func callback function.

### Observations

Run the project and see the same behavior as in the previous project. The only exception is the new explosion effects after the initial project start. Notice how each explosion persists for a short while before disappearing gradually. Comparing this effect with the one resulting from a short tapping of the Q key, observe that without a dedicated particle emitter, the explosion seems to have fizzled before it begins.

**Other type of emitter, e.g., controlled by time, emits forever, number emiting over time defined by some function, emitter should be able to attach to geometry (GameObject, think rocket)**

# Summary

In addition, you have integrated a basic particle system with your physics engine with particles that interact with the RigidShape objects in the system. Through working with your particle system, you have learned that appropriate use of randomness is important and that creating interesting visual effects requires hands-on experience and fine-tuning iterations.

## Game Design Considerations