Creating Effects with Particle Systems

After completing this chapter, you will be able to:

* Understand the fundamentals of a particle, a particle emitter, and a particle system
* Appreciate that many interesting physical effects can be modelled base on a collection of dedicated particles
* Approximate the basic behavior of a particle such that the rendition of a collection of these particles resemble a simple explosion-like effect
* Implement a straightforward particle system that is integrated with the RigidShape system of the physics component

# Introduction

So far in your game engine it is assumed that the game world can be described by a collection of geometries where all objects are Renderable instances with texture, or animated sprite, and potentially illuminated by light sources. This game engine is powerful and capable of describing a significant portion of objects in the real-world. However, it is also true that it can be challenging for your game engine to describe many everyday encounters. For example, sparks, fire, explosions, dirt, dust, etc. Many of these observations are transient effects resulting from matters changing physical states, or a collection of very small size entities reacting to physical disturbances. Collectively, these observations are often referred to as special effects and in general do not lend themselves well to being represented by fixed-shape geometries with textures.

Particle systems describe special effects by emitting a collection of particles with properties that may include: position, size, color, life-time, and strategically selected texture maps. These particles are defined with specific behaviors where once emitted, their properties are updated to simulate a physical effect. For example, a fire particle may be emitted to move in an upward direction with reddish color. As time progresses, the particle may decrease in size, slow the upward motion, change its color towards yellow, and eventually disappear after certain number of updates. With strategically designed update functions, the rendition of a collection of such particles can resemble a fire burning.

In this chapter, you will study, design, and, 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 integrate your particles within your scenes. The particles will collide and interact accordingly with the RigidShape objects. You will also discover the need for and define particle emitters to generate particles over a period of time such as a campfire or torch.

The main goal of this chapter is to understand the fundamentals of a particle system: attributes and behaviors of simple particles, details of a particle emitter, and the integration with the rest of the game engine. This chapter does not lead you to create any specific types of special effects. This is analogues to learning an illumination model in Chapter 8 without the details of creating any lighting effects. The manipulation of light source parameters and material properties to create engaging lighting conditions, and the modeling of particle behaviors that resemble specific physical effects are the responsibilities of the game developers. The basic responsibility of the game engine is to define sufficient fundamental functionality to ensure that the game developers can accomplish their job.

# Particles and Particle Systems

A particle is a textured position without 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 an area. The important clarification is that the game engine logic processes a particle as a position with no area, while the drawing system displays the particle as a texture with proper dimensions. 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.

In addition to a position, a particle also has properties such as size (for scaling the texture), color (for tinting the texture), and life span. Similar to a typical game object, each particle is defined with behaviors that modify its properties during each update. It is the responsibility of this update function to ensure that the rendition of a collection of particles resembles a familiar physical effect. A particle system is the entity that controls the spawning, updating, and removal of each individual particles. In your game engine, particle systems will be defined as a separate component, just like the physics component.

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, including custom GLSL shaders, default sharable shader instance, and a new 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

This project is a continuation from the previous chapter and supports all of the rigid shape and collision controls. For brevity the details of those controls will not be restated in this chapter. The particle system specific controls of the project are as follows:

* Q key: To spawn particles at the current mouse position
* E key: To toggle the drawing of particle bounds

The goals of the project are as follows:

* To understand the details of how to draw a particle and define its behavior
* To implement a simple 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 particle.png, the default particle texture, and the same four texture images from previous projects:

* minion\_sprite.png defines the sprite elements for the hero and the minions
* platform.png defines the platforms, floor, and ceiling tiles;
* wall.png defines the walls
* target.png identifies the currently selected object.

### Supporting Drawing of a Particle

Particles are textured positions with no area. However, as discussed in the introduction, 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.

#### Creating GLSL Particle Fragment Shader

When it comes to the actual computation of each pixel color, a new GLSL fragment shader, particle\_fs.glsl, must be created to ignore the global ambient terms. Physical effects such as fires and explosions do not participate in illumination computations.

1. Under the src/glsl\_shaders folder, create 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 without considering global ambient effect. This serves as one approach for computing the colors of the particles. This function can be modified to support different kinds of particle effects.

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 define a default particle shader instance to be shared. Recall from working with other types of shaders in the previous chapters that shaders are created once and shared engine-wide in the shader\_resoruces.js file in the src/engine/core folder.

1. Begin by editing the shader\_resources.js file in the src/engine/core folder to define the constant, variable, and accessing function for the default particle shader.

// Particle Shader

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

let mParticleShader = null;

function getParticleShader() { return mParticleShader }

1. In the init() function, make sure to load the newly defined particle\_fs GLSL fragment shader.

function init() {

let loadPromise = new Promise(

async function(resolve) {

await Promise.all([

… identical to previous code …

text.load(kShadowReceiverFS),

text.load(kParticleFS)

]);

resolve();

}).then(

function resolve() { createShaders(); }

);

map.pushPromise(loadPromise);

}

1. With the new GLSL fragment shader, particle\_fs, properly loaded, you can instantiate a new particle shader when the createShaders() function is called.

function createShaders() {

… identical to previous code …

mShadowReceiverShader = new SpriteShader(kTextureVS, kShadowReceiverFS);

mParticleShader = new TextureShader(kTextureVS, kParticleFS);

}

1. In the cleanUp() function remember to perform the proper cleanup and unload operations.

function cleanUp() {

… identical to previous code …

mShadowCasterShader.cleanUp();

mParticleShader.cleanUp();

… identical to previous code …

text.unload(kShadowReceiverFS);

text.unload(kParticleFS);

}

1. Lastly, do not forget to export the newly defined function.

export {init, cleanUp,

getConstColorShader, getTextureShader, getSpriteShader, getLineShader,

getLightShader, getIllumShader, getShadowReceiverShader, getShadowCasterShader,

getParticleShader}

#### Creating the ParticleRenderable Object

Recall that a SimpleShader/Renderable pair of objects must be defined to interface the GLSL shader to the game engine. With the default particle shader object defined to interface to the GLSL particle\_fs shader, you can now create a new Renderable object type 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.

In the src/engine/renderables folder, create the particle\_renderable.js file, import from defaultShaders for accessing the particle shader and from TextureRenderable for the base class. Define the ParticleRenderable to be a subclass of TextureRenderable, and set the proper default shader in the constructor. Remember to export the class.

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;

#### Loading the Default Particle Texture

For convenience when drawing, the game engine will preload the default particle texture, particle.png, located in the assets/particles folder. This operation can be integrated as part of the defaultResources initialization process.

1. Edit default\_resources.js in the src/engine/resources folder, add an import from texture.js to access the texture loading functionality, define a constant string for the location of the particle texture map, and an accessor for this string.

import \* as font from "./font.js";

import \* as texture from "../resources/texture.js";

import \* as map from "../core/resource\_map.js";

// Default particle texture

let kDefaultPSTexture = "assets/particles/particle.png";

function getDefaultPSTexture() { return kDefaultPSTexture; }

1. In the init() function, call the texture.load() function to load the default particle texture map.

function init() {

let loadPromise = new Promise(

async function (resolve) {

await Promise.all([

font.load(kDefaultFont),

texture.load(kDefaultPSTexture)

]);

resolve();

})

… identical to previous code …

}

1. In the cleanUp() function, make sure to unload the default texture.

function cleanUp() {

font.unload(kDefaultFont);

texture.unload(kDefaultPSTexture);

}

1. Finally, remember to export the accessor.

export {

… identical to previous code …

getDefaultFontName, getDefaultPSTexture,

… identical to previous code …

}

With this integration, the default particle texture file will be loaded into the resource\_map during system initialization. This default texture map can be readily accessed with the returned value from the getDefaultPSTexture() function.

### Defining the Engine Particle Component

With the drawing infrastructure defined, you can now define the engine component to manage the behavior of the particle system. For now, the only functionality required is to include a default system acceleration for all particles.

In the src/engine/components folder, create the particle\_system.js file and define the variable, getter, and setter functions for the default particle system acceleration. Remember to export the newly defined functionality.

let mSystemAcceleration = [30, -50.0];

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

function setSystemAcceleration(x, y) {

mSystemAcceleration[0] = x;

mSystemAcceleration[1] = y;

}

export {getSystemAcceleration, setSystemAcceleration}

Before continuing, make sure to update the engine access file, index.js, to allow game developer access to the newly defined functionality.

### Defining the Particle and Particle Game Classes

You are now ready to define the actual particle, its default behaviors, and the class for a collection of particles.

#### Creating a Particle

Particles are lightweight game objects with simple properties wrapping around ParticleRenderable for drawing. To properly support motion, particles also implement movement approximation with the Symplectic Euler Integration.

1. Begin by creating the particles subfolder in the src/engine folder. This folder will contain particle specific implementation files.
2. In the src/engine/particles folder create particle.js, and define the constructor to include variables for position, velocity, acceleration, drag, and drawing parameters 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;

}

… implementation to follow …

}

export default Particle;

1. Define the draw() function to draw the particle as a TextureRenderable, and a drawMarker() debug function to draw an X marker at the position of the particle.

draw(aCamera) {

this.mRenderComponent.draw(aCamera);

}

drawMarker(aCamera) {

let size = this.getSize();

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

}

1. You can now implement the update() function to compute the position of the particle based on Symplectic Euler Integration, where the scaling with the mDrag variable simulates drags on the particles. Notice that this function also performs incremental changes to the other parameters including color and size. The mCyclesToLive variable informs the particle system when it is appropriate to remove this particle.

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 ParticleSet

To work with a collection of particles, you can now create the ParticleSet to support convenient looping over sets of Particle. For lightweight purposes, the Particle class does not subclass from the more complex GameObject, however, as JavaScript is an untyped language, it is still possible for ParticleSet to subclass from and refine GameObjectSet to take advantage of the existing set-specific functionality.

1. In the src/engine/particles folder, create particle\_set.js, and define the 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();

}

… implementation to follow …

}

export default ParticleSet;

1. Override the draw() function of GameObjectSet to ensure particles are drawn with additive blending.

**Note** Recall from Chapter 5 that the default gl.blendFunc() setting implements transparency by blending the alpha channel values. This is referred to as alpha blending. In this case, the gl.blendFunc() setting simply accumulates colors without considering the alpha channel. This is referred to as additive blending. Additive blending often results in oversaturation of pixel colors, i.e., RGB components with values of greater than the maximum displayable value of 1.0. The oversaturation of pixel color is often desirable when simulating intense brightness of fire and explosions.

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);

}

}

}

Lastly, remember to update the engine access file, index.js, to forward the newly defined functionality to the client.

### 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 by being properly created, destroyed and behaving as expected. The test case is based mainly on the previous project with a new \_createParticle() function that is called when the Q key is pressed. The \_createParticle() function implemented in the my\_game\_main.js file creates particles with pseudo random behaviors as listed in the following.

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;

}

There are two important observations to be made on the \_createParticle() function. First, the random() function is used many times to configure each created Particle. Particle systems utilize large numbers of similar particles with slight differences to build and convey the desired visual effect. It is generally important to avoid any patterns by using randomness. 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 is unfortunately the nature of working with particle systems. There is often quite a bit of ad hoc experimentation. 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 adjusting the provided presets.

### 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 as though there is an engine generating flames beneath the mouse. Type the E key to toggle the drawing of individual particle positions. Now you can observe a green-X marking the position of each of the generated particles.

If you move the mouse pointer rapidly, you can observe individual pink circles with green-X centers changing color while dropping towards the floor. Although all particles are created by the \_createParticle() function and share the similar behaviors of falling towards the floor while changing color, every particle appears slightly different and does not exhibit any behavior patterns. You can now clearly observe the importance of integrating randomness in the created particles.

There are limitless variations to how you can modify the \_createParticle() function. For example, you can change the explosion-like effect to steam or smoke simply by changing the initial and final color to different shades of grey and transparencies. Additionally, you can modify the default particle texture by inverting the color to create black smoke effects. You could also modify the size change value, delta to that of greater than 1 to increase the size of the particles over time. There are literally no limits to how particles can be created. The particle system you have implemented allows the game developer to create particles with customized behaviors that are most suitable to the game that they are building.

Lastly, notice that the generated particles do not interact with the RigidShape objects and appears as though the particles are drawn over the rest of the objects in the game scene. This issue will be examined and resolved in the next project.

# Particle Collisions

An approach to integrate particles into a game scene is for the particles to follow the implied rules of the scene and interact with the non-particle objects accordingly. The ability to detect collisions is the foundation for interactions between objects. For this reason, it is sometimes important to support particle collisions with the other, non-particle game objects.

Since particles are defined only by their positions with no dimensions, the actual collision computations can be relatively straightforward. However, there are typically a large number of particles, as such, the number of collisions to be performed can also be numerous. As a compromise and optimization in computational costs, particles collisions can be based on RigidShape instead of the actual Renderable objects. This is similar to the case of the physics component where the actual simulation is based on simple rigid shapes in approximating the potentially geometrically complicated Renderable objects.

## The Particle Collisions Project

This project demonstrates how to implement a particle collision system that is capable of resolving collisions between particles and the existing RigidShape objects. 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.particle\_collisions folder.



Figure 10-2. Running the Particle Collisions project

The controls of the project are identical to the previous project and supports all of the rigid shape and collision controls. The controls that are specific to the particle system are as follows:

* Q key: To spawn particles at the current mouse position
* E key: To toggle the drawing of particle bounds
* 1 key: To toggle Particle/RigidShape collisions

The goals of the project are as follows:

* To understand and resolve collisions between individual positions and RigidShape objects
* To build a particle engine component that supports interaction with RigidShape

### Modifying the Particle System

With a well-designed infrastructure, implementation of new functionality can be localized. In the case of particle collisions, all modifications are within the particle\_system.js file in the src/engine/components folder.

1. Edit particle\_system.js to define and initialize temporary local variables for resolving collisions with RigidShape objects. The mCircleCollider object will be used to represent individual particles in collisions.

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();

}

1. Define the resolveCirclePos() function to resolve the collision between a RigidCircle and a position by pushing the position outside of the circle shape.

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;

}

1. Define the resolveRectPos() function resolve the collision between a RigidRectangle and a position by wrapping the mCircleCollider local variable around the position and invoking the RigidCirlce to RigidRectangle collision function. When interpenetration is detected, the position is pushed outside of the rectangle shape according to the computed mCollisionInfo.

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;

}

1. Implement resolveRigidShapeCollision() and resolveRigidShapeSetCollision() to allow convenient invocation by client game developers. These functions resolve collisions between a single and a set of RigidShape and a ParticleSet objects.

// 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;

}

1. Lastly, remember to export the newly defined functions.

export {init,

getSystemAcceleration, setSystemAcceleration,

resolveRigidShapeCollision, resolveRigidShapeSetCollision}

### Initializing the Particle System

The temporary variables defined in particle\_system.js must be initialized before the game loop begins. Edit loop.js, import from particle\_system.js, and call the init() function after asynchronous loading is completed in the start() function.

… identical to previous code …

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

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

… identical to previous code …

async function start(scene) {

… identical to previous code …

// Wait for any async requests before game-load

await map.waitOnPromises();

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

particleSystem.init();

… identical to previous code …

}

### Testing the Particle System

The modifications required for the MyGame class are straightforward. A new variable must be defined to support the toggling of collision resolution, and the update() function defined in my\_game\_main.js is modified as follows.

update() {

… identical to previous code …

if (engine.input.isKeyClicked(engine.input.keys.One))

this.mPSCollision = !this.mPSCollision;

if (this.mPSCollision) {

engine.particleSystem.resolveRigidShapeSetCollision(this.mAllObjs, this.mParticles);

engine.particleSystem.resolveRigidShapeSetCollision(this.mPlatforms, this.mParticles);

}

… identical to previous code …

}

### Observations

As in previous projects, you can run the project and create particles with the Q and E keys. However, notice that the generated particles do not overlap with any of the objects. You can even try moving your mouse pointer to within the bounds of one of the RigidShape objects and then type the Q key. Notice that in all cases, the particles are generated outside of the shapes.

You can try typing the 1 key to toggle collisions with the rigid shapes. Note that with collisions enabled, the particles somewhat resemble the amber particles from a fire or an explosion where they bounce off the surfaces of RigidShape objects in the scene. When collision is toggled off, as you have observed from the previous project, the particles appear to be burning or exploding in front of the other objects. In this way, collision is simply another parameter for controlling the integration of the particle system with the rest of the game engine.

You may find it troublesome to continue to press on the Q key to generate particles. In the next project, you will learn about generation of particles over a fixed period of time.

# Particle Emitters

With your current particle system implementation, you can create particles at a specific point and time. These particles 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 or firework that persists for a short while after the creation of a new RigidShape object. 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 particle emission 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 identical to the previous project and supports all of the rigid shape and collision controls. The particle system specific controls of the project are as follows:

* Q key: To spawn particles at the current mouse position
* E key: To toggle the drawing of particle bounds
* 1 key: To toggle Particle/RigidShape collisions

The goals of the project are as follows:

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

### Defining the ParticleEmitter Class

You have observed and experienced the importance of avoiding patterns when working with particles. In this case, as the ParticleEmitter object generates new particles over time, it is again important to inject randomness to avoid any appearance of a pattern.

1. In the src/engine/particles folder, create particle\_emitter.js, define the ParticleEmitter class with a constructor that receives the location, number, and how to emit new particles. Note that the mParticleCreator variable expects a callback function. When required, this function will be invoked to create a particle.

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;

}

… implementation to follow …

}

export default ParticleEmitter;

1. Define a function to return the current status of 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 create or 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, it is unlikely to encounter 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 the actual behavior of each particle.

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);

}

}

Lastly, remember to update the engine access file, index.js, to allow game developer access to the ParticleEmitter class.

### Modifying the Particle Set

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

1. Edit particle\_set.js in the src/engine/particles folder, and define a new variable for maintaining emitters.

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 and to remove expired emitters.

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);

}

}

}

### Testing the Particle Emitter

This is a straightforward test of the correct functioning of the ParticleEmitter object. The MyGame class update() function is modified to create a new ParticleEmitter at the position of the RigidShape object when the G or H key is pressed. In this way, it will appear as though an explosion has occurred when a new RigidShape object is created, or, when RigidShape objects are assigned new velocities.

In both cases, the \_createParticle() function discussed in the first project of this chapter is passed as the argument for the createrFunc callback function parameter in the ParticleEmitter constructor.

### Observations

Run the project and observe the initial firework-like explosions at the locations where the initial RigidShape objects are created. Type the G key to observe the accompanied explosion in the general vicinity of the newly created RidigShape object. Alternatively, you can type the H key to apply velocities to all the shapes and observe explosion-like effects next to each RigidShape object. For a very rough sense of what this particle system may look like in a game, you can try enabling texturing (with the T key), disabling RigidShape drawing (with the R key), and typing the H key to apply velocities. Observe that it appears as though the Renderable objects are being blasted by the explosions.

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, and observe that without a dedicated particle emitter, the explosion seems to have fizzled before it begins.

Similar to particles, emitters can also have drastically different characteristics for simulating different physical effects. For example, the emitter you have implemented is driven by the number of particles to create. This behavior can be easily modified to use time as the driving factor, e.g., emitting an approximated number of particles over a given time period. Other potential applications of emitters can include, but are not limited to:

* allowing the position of the emitter to change over time, e.g., attaching the emitter to the end of a rocket
* allowing emitter to affect the properties of the created particles, e.g., changing the acceleration or velocity of all created particles to simulate wind effects

Based on the simple and yet flexible particle system you have implemented, you can now experiment with all these ideas in a straightforward manner.

# Summary

There are three simple takeaways from this chapter. First, you have learned that particles, positions with an appropriate texture and no dimensions, can be useful in describing interesting physical effects. Second, the capability to collide and interact with other objects assists with the integration and placement of particles in game scenes. Lastly, in order to achieve the appearance of familiar physical effects, the emitting of particles should persist over some period of time.

You have developed a simple and yet flexible particle system to support the consistent management of individual particles and their emitters. Your system is simple because it consists of a single component, defined in particle\_system.js, with only three simple supporting classes defined in the src/engine/particles folder. The system is flexible because of the callback mechanism for the actual creation of particles where the game developers are free to define and generate particles with any arbitrary behaviors.

The particle system you have built serves to demonstrate the fundamentals. To increase the sophistication of particle behaviors, you can subclass from the simple Particle class, define additional parameters, and amend the update() function accordingly. To support additional physical effects, you can consider modifying or sub-classing from the ParticleEmitter class and emit particles according to your dsired formulations.

## Game Design Considerations

As discussed in Chapter 9, presence in games isn’t exclusively achieved by recreating our physical world experience in game environments; while introducing real-world physics is often an effective way to bring players into virtual worlds there are many other design choices that can be quite effective at pulling players into the game, either in partnership with object physics or on their own. For example, imagine a game with a 2D comic-book visual style that displays a “BOOM!” text-based image whenever something explodes; objects don’t show the word “BOOM!” when they explode in the physical world, of course, but the stylized and familiar use of “BOOM!” in the context of a comic book visual aesthetic can be quite effective on its own as a way to connect players with what’s happening in the game world.



Figure 10-4. Visual techniques like those shown in this graphic are often used in graphic novels to represent various fast-moving or high-impact actions like explosions, punches, crashes, and the like; similar visual techniques have also been used quite effectively in film and video games.

Particle effects can also be used either in realistic ways that mimic how we’d expect them to behave in the real world or in more creative ways that have no connection to real-world physics. Try using what you’ve learned from the examples in this chapter and experiment with adding particles to your current game prototype as we left it in Chapter 9: can you think of some uses for particles in the current level that might support and reinforce the presence of existing game elements (e.g., sparks flying if the player character touches the force field)? What about introducing particle effects that might not directly relate to game play but enhance and add interest to the game setting?