Урок 4

Оптимизация проекта

Оптимизация работы с памятью. Управление объектами через Emitter-классы. Использование паттерна Object Pool. Использование атласов текстур.

Оптимизация работы с памятью

При создании объектов для них выделяется память в куче. Как только куча переполняется, сборщик мусора очищает неиспользуемые объекты, чтобы избежать «OutOfMemoryError». Во время сборки мусора может наблюдаться значительное снижение производительности, что является крайне нежелательным в процессе игры. Чтобы исключить такие просадки производительности, следует избегать лишнего создания объектов, насколько это возможно.

Например, метод Vector2.cpy() создает новый экземпляр типа Vector2, вместо того, чтобы повторно использовать существующий экземпляр, что при 60 кадрах в секунду будет приводить к созданию 60 новых объектов типа Vector2 каждую секунду. Это заставит сборщик мусора срабатывать чаще.

Работа с ресурсами

Игры используют довольно тяжелые ресурсы, вроде изображений и звуковых эффектов, которые могут занимать значительное количество оперативной памяти. Кроме того, сборщик мусора Java не управляет большинством этих ресурсов. Вместо этого они управляются посредством собственных драйверов.

Для оптимизации работы с памятью в LibGDX есть классы, которые представляют такие ресурсы. Все они реализуют общий Disposable-интерфейс, который указывает, что экземпляры класса должны быть удалены вручную в конце времени жизни. Для этого у них необходимо вызвать метод dispose().

Не освобождение ресурсов (захламление) приведет к серьезным утечкам памяти. Кроме того, доступ к ресурсу, который уже удален, приведет к неопределенным ошибкам, поэтому убедитесь, что все ссылки, указывающие на удаленные ресурсы, очищены.

Следующие классы LibGDX должны быть удалены вручную: AssetManager, Bitmap, BitmapFont, BitmapFontCache, CameraGroupStrategy, DecalBatch, ETC1Data, FrameBuffer, Mesh, ParticleEffect, Pixmap, PixmapPacker, ShaderProgram, Shape, Skin, SpriteBatch, SpriteCache, Stage, Texture, TextureAtlas, TileAtlas, TileMapRenderer, World.

Ресурсы должны быть удалены, как только они больше не нужны, освобождая связанную с ними память.

Пулы объектов

Пулы объектов — принцип повторного использования неактивных или «мертвых» объектов, вместо постоянного создания новых. Это достигается с помощью создания пула объектов, и когда вам нужен новый объект, вы получаете его из этого пула. Если в пуле есть свободный объект, он его вернет. Если пул пустой или не содержит свободных объектов, возвращается созданный новый экземпляр объекта. Когда объект больше не нужен, вы освобождаете его: это означает его возвращение обратно в пул. Таким образом, повторно используется память для размещения объектов и сборщик мусора просто «счастлив». Это жизненно важно для управления памятью в игре, которая часто порождает такие объекты, как пули, препятствия, монстры и т. д.

LibGDX предоставляет интерфейсы и классы для работы с пулом: Poolable, Pool, Pools.

Реализация Poolable-интерфейса означает, что объект имеет метод reset(), который в LibGDX автоматически вызывается при освобождении объекта. Ниже приведен класс Bullet, реализующий интерфейс Pool.Poolable:

```
public class Bullet implements Pool.Poolable {
    Vector2 position;
    Vector2 velocity;
    boolean active;

    public Bullet() {
        this.position = new Vector2(0, 0);
        this.velocity = new Vector2(0, 0);
        this.active = false;
    }

    public void setup(float x, float y, float vx, float vy) {
        position.set(x, y);
        velocity.set(vx, vy);
        active = true;
    }

    @Override
    public void reset() {
        active = false;
        position.set(0, 0);
    }

    public void destroy() {
        active = false;
    }

    public void update(float dt) {
        position.mulAdd(velocity, dt);
        if (position.x < -20 || position.x > 1300 || position.y < -20 ||
position.y > 740) {
            destroy();
        }
    }
}
```

Теперь управляем пулями в классе «BulletEmitter» через пул объектов:

```
public class BulletEmitter {
    private static final BulletEmitter ourInstance = new BulletEmitter();

    public static BulletEmitter getInstance() {
        return ourInstance;
    }
```

```
    final Array<Bullet> activeBullets = new Array<Bullet>();

    final Pool<Bullet> bulletsPool = new Pool<Bullet>(256, 8192) {
        @Override
        protected Bullet newObject() {
            return new Bullet();
        }
    };

    TextureRegion texture;

    public void reset() {
        bulletsPool.clear();
        activeBullets.clear();
    }

    private BulletEmitter() {
        texture = Glob.getInstance().assetManager.get("my.pack",
TextureAtlas.class).findRegion("bullet");
    }

    public void update(float dt) {
        Bullet bullet;
        int len = activeBullets.size;
        for (int i = len; --i >= 0; ) {
            bullet = activeBullets.get(i);
            bullet.update(dt);
            if(!bullet.active) {
                activeBullets.removeIndex(i);
                bulletsPool.free(bullet);
            }
        }
    }

    public void render(SpriteBatch batch) {
        Bullet bullet;
        int len = activeBullets.size;
        for (int i = len; --i >= 0; ) {
            bullet = activeBullets.get(i);
            batch.draw(texture, bullet.position.x - 16, bullet.position.y - 16);
        }
    }

    public void setupBullet(Vector2 position, float angle) {
        setupBullet(position.x, position.y, angle);
    }

    public void setupBullet(float x, float y, float angle) {
        Bullet item = bulletsPool.obtain();
```

```
        item.setup(x, y, 400 * (float) cos(angle), 400 * (float) sin(angle));
        activeBullets.add(item);
    }
}
```

Класс Pools предоставляет статические методы для динамического создания пулов любых объектов, реализующих интерфейс Poolable.

Упаковщик текстур

В OpenGL привязывается текстура и делается визуализация, затем привязывается другая текстура и выполняется дополнительная визуализация, и так далее. Привязывание текстуры является дорогой операцией, поэтому рекомендуется хранить большое число маленьких изображений на одном большом. Таким образом, выполняется привязывание один раз большой текстуры, а затем идет многократная визуализация отдельных частей.

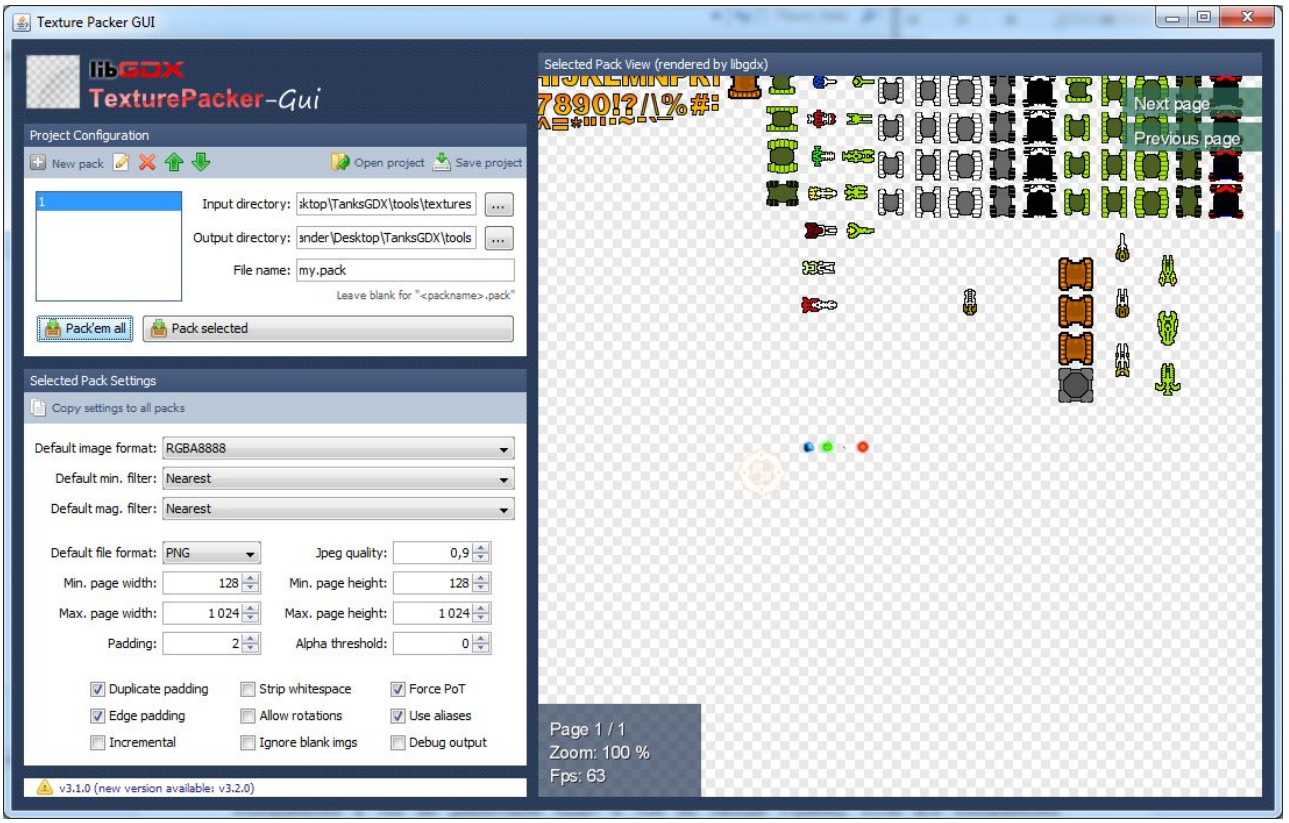
LibGDX имеет класс TexturePacker, который представляет собой приложение командной строки для упаковки множества маленьких изображений в одно большое. Он хранит местоположение маленьких изображений, так что в приложении на них можно ссылаться по имени, используя класс TextureAtlas. TexturePacker находится в проекте gdx-tools и может быть запущен из исходного кода:

```
import com.badlogic.gdx.tools.texturepacker.TexturePacker;

public class MyPacker {
    public static void main (String[] args) throws Exception {
        TexturePacker.process(inputDir, outputDir, packFileName);
    }
}
```

Если класс TexturePacker не найден, добавьте gdx-tools в build.gradle-файл.

Другой вариант упаковки текстур – использование инструмента с графическим интерфейсом, который можно скачать по адресу: https://code.google.com/archive/p/libgdx-texturepacker-gui/downloads.



В программе указывается папка с текстурами (Input Directory), папка, куда будет сохранен атлас (Output directory) и имя самого атласа (File name).

Структура директорий

TexturePacker может запаковать все изображения для приложения за один раз. Указывая директорию, он будет рекурсивно сканировать файлы изображений. Для каждой директории с изображениями он запаковывает их в одну большую текстуру, называемую *страницей*. Если изображения в директории не помещаются в максимальный размер одной страницы, используется несколько страниц.

Изображения в той же директории будут в том же наборе страниц. Если все изображения помещаются на одну страницу, тогда не будут использоваться поддиректории, потому что с одной страницей приложение выполнит только одну привязку текстуры. В противном случае поддиректории могут быть использованы для разделения взаимосвязанных изображений и уменьшения привязок текстур.

Например, приложение может «захотеть» поместить все игровые изображения в отдельную директорию от изображений для меню паузы. Так как существует два набора изображений, то визуализируются они последовательно: сначала визуализируются все игровые изображения (одна привязка), затем меню паузы визуализируется сверху (другая привязка). Если изображения разместить в одной директории (что приведет к более чем одной странице), каждая страница будет содержать смесь игровых изображений и меню паузы. Это может привести к нескольким привязкам текстур для визуализации игры и меню паузы вместо одной привязки для соответствующих изображений.

TextureAtlas

TexturePacker выдает директорию страниц изображений и текстовый файл, описывающий все изображения, запакованные в страницы. Ниже показано, как использовать изображения в приложении:

```
TextureAtlas atlas;
atlas = new TextureAtlas(Gdx.files.internal("packedimages/pack.atlas"));
AtlasRegion region = atlas.findRegion("ship");
```

TextureAtlas читает файл упаковки и загружает все изображения страницы. Можно получить TextureAtlas.AtlasRegion, являющиеся подклассом TextureRegion, которые предоставляют дополнительную информацию о запакованном изображении: индекс кадра или пустую область, которая была удалена. Метод findRegion() не очень быстрый, так что следует хранить возвращаемое значение вместо вызова метода при каждом кадре. TextureAtlas содержит все текстуры страниц. Высвобождение TextureAtlas высвободит все текстуры страниц.

Домашнее задание

1. Вы найдете его на странице занятия.

Дополнительные материалы

1. https://github.com/libgdx/libgdx/wiki/Memory-management
2. https://github.com/libgdx/libgdx/wiki/Texture-packer

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. http://www.libgdx.ru/2015/01/texture-packer.html
2. https://github.com/libgdx/libgdx/wiki/Texture-packer
3. https://github.com/libgdx/libgdx/wiki/Memory-management