39

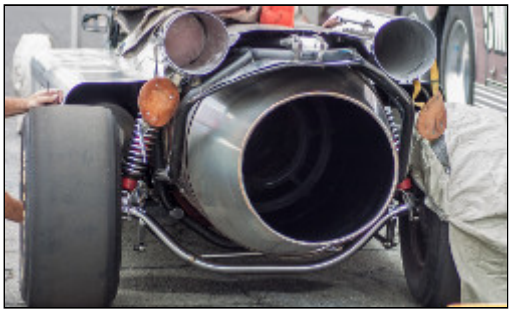
Making a Game Engine: Core Design Principles

12

By **James Lambert** | Published Jan 29 2014 04:50 PM in Game Programming
Peer Reviewed by (Servant of the Lord, Michael Tanczos, dejaime)

architecture | game engine

See Also:
Making a Game Engine: Transform Hierarchy



Before I get started I want to say a few things. Typically, you don't want to make a game engine, you want to make a game. I hope that this article can help you work on some core design concepts that you can apply when making your game. If you use good architecture you will find your game is less buggy and much easier to maintain. You will also find that you will be able to reuse a lot of code between projects. This reusable core becomes your game engine. Now before I talk about some game-engine-specific design I want to discuss some good design principles for programming in general.

General Programming Concepts

While the field of software architecture is vast and volumes of books could be written on the subject, I have chosen to focus on a few important principles that I find to be very important to making a solid game engine. This article alone does not intend to be enough to make a solid core for you game. That is a vast topic and would make a very large article if put in one place but I do hope to help you get one step closer to your goal of completing a game.

Cohesion

A highly cohesive class is one that has one and only one clear purpose. I think the best way to understand this is to give a bad example of cohesion then suggest ways to fix it.

```
class Game
{
        Player player;
    Enemy[] enemies;

    void update()
    {
        foreach (Enemy enemy in enemies)
        {
                if (Math.abs(enemy.y - player.y) < EnemySightRange && Math.abs(ene
            {
                enemy.moveTowards(player.x, player.y);
            }
        }

        if (input.keyDown('left'))
        {
                player.move(-1, 0);
        }
        // ... rest of player logic
    }

    void draw()
    {
        player.draw();
        foreach (Enemy enemy in enemies)
        {
                enemy.draw();
        }
    }
}
```

So let me point out a few things wrong with this design. First of all, the class `Game` has knowledge that there is a single player and a list of enemies. Why is this a problem? Because this class is not reusable nor is it very flexible. Let's say we were using this code and decided we wanted to make the game multiplayer. With the current code, we have single player hard-coded into the game class so adding another would likely be a difficult and buggy procces. Also, the enemy game logic is inside the game class as well. It doesn't belong there.

Another subtle problem with this design is that both the player and enemy directly have an `x` and `y` variable. This x, y position should be grouped together and the player's position should be accessed using `player.position` or even `player.transform.position` where the transform of the player contains information about location, rotation, and size of the player.

The last thing I want to point out is that the game class directly draws the players and enemies and that the player and enemies each have draw methods as well. The problem here is the way the player or enemy is drawn is linked

directly with all other attributes of the player. This may not seem like such a problem on the surface but can create problems as things get more complex. It is better to remove drawing code from the player class and putting it into a seperate class.

So when writing your games how can you avoid low cohesion and classes and know when you need to break things up? The best way is to describe what that class does in words. If you cannot do it in one sentence without using the word 'and' then you probably should break up the functionality of the class. In the example above the responsibilities of the game class are updating the enemies *and* updating the player *and* drawing the player *and* enemies. So how do you fix this? First of all, create a generic game object and add components to it that each have a specific task. One component to render it, another to handle game logic, another for physics, (etc...). This is far better than making sublcasses of a game object class to add custom behaviour. More on this later.

Coupling

Coupling refers to classes using other classes. An example would be a player class containing a weapon class. In this case the player is coupled to the weapon. To promote good design you want to have classes as decoupled as possible. You cannot, of course, remove all coupling between classes, because at some point one class will have to you use another class, but you want have as few dependencies as possible. So again I am going to give a poor example and show why it is less than ideal.

```
class PlayerRenderer
{
        Player player;
    bool isFlashing;

        void draw()
    {
        if (Player.collider.isTouchingEnemy())
        {
                this.flash();
        }

        // draw the player
    }
}
```

So this class also has some cohesion problems: the renderer is responsible for colliding against enemies and flashing the player, but I want to focus on the coupling problems. First of all, There is a player renderer. The name of the class `PlayerRenderer` implies that this class depends on the player but it is also a render system. This class also has a reference to the player collider in it. This means that the drawing system is no longer independent from the collision system and the game logic. Now, changes in the collision system or on the player can effect the render code. Another issue is that the collider class has a `isTouchingEnemy` method. This means that the collider knows about the enemies in the game. When a program is tightly coupled like this bugs in one part of the system can propagate and cause problems in seemingly unrelated places. By removing theses inter-dependencies you make it much easier to make changes in one part of the code without worrying about the rest. You could change entirely the rendering engine without touching the collider code. Why would this be useful? Well, if you wanted to make a game engine to support both DirectX and OpenGL all that needs to change is the render module. The same thing applies with porting games to consoles. All of the game logic can stay the same, all that needs to change is the rendering system, input sytem, and any other hardware-related systems.

How do you fix the problem in the code above? By making a generic renderer and a generic collider. These classes should have no knowledge about what type of object is using them. You should just be able to give them parameters, such as what image to draw and other parameters such as color and position and the sub system handles the rest. You then write a class that can interact with these different subsystems and orchestrate them together into a single

working game. More on the subject later.

The Singleton

Singletons are evil. Don't use them.

To elaborate on this more, the use of singletons often creates problems for maintaining the code. They create implicit dependencies in the code and often lead to poor coupling. Since the singleton is accessible anywhere in the code, many times it is used everywhere in the code. If at any time you decide you don't want to have only a single instance of a class you are out of luck. The singleton is already all over the code and it will be a difficult process removing it. Another reason, and possibly one of the the biggest for not using singletons, is that they are often used because they are quick and easy and don't require you to think about the design. Meaning you don't have to carefully plan out your code. You simply make all of the major systems a singleton and allow any object to access anything else at any point. This usually ends up putting code where it doesn't belong and makes certain parts of the code do things it shouldn't be responsible for. When I stopped using singletons I found that I was able to come up with much better design decisions in my code. Because I thought more about how variables and objects would be passed around, it forced me to assess my design and think about it more. Do yourself a favor and don't use singletons.

Game Engine Design

So now that we have discussed some general design concepts lets focus on an example of how to structure the core of your game. Keep in mind that the method I am going to describe is not the only way to write a game engine. There is no one right way to make a game.

Game Loop

Before I get into game objects and scene I wanted to mention the game loop. Simply put, a game loop first updates the game, then draws a frame and continuously repeats those two things. The game loop should regulate the frame rate to match the refresh rate of the monitor and also compensate with larger time steps when the frame rate begins to drop. I am not going to spend any more time on this subject but here is an excellent article on the game loops.

Scene

The scene is what manages all of the game objects. It should receive update and draw messages and pass those onto the components attached to game objects in the scene. A scene represents a level in a game. You can add terrain, enemies, objects, lights, cameras and anything else that belongs in games into the scene. The structure of a scene may look something like this.

```
class Scene
{
        GameObjects[] gameObjects;
        EventDispatcher eventDispatcher;
        SceneRenderer sceneRenderer;

        void update(float timestep, InputState input)
        {
                eventDispatcher.sendEvent("update", timestep, input);
        }

        void render()
        {
                eventDispatcher.sentEvent("render", sceneRenderer);
        }

        void createGameObject()
        {
                GameObject result = new GameObject();
                gameObjects.add(result);
                result.addedToScene(this);

                // add any component in the gameObject to the eventDispatcher
                // so it can receive update and render messages
                // if the component doesn't have either method it isn't added
        }
}
```

Now the scene is completely decoupled from the specifics of what kind of game this is. This core scene class can be reused in any number of game types and serves as part of the foundation of the game engine. A scene can also contain addons, such as the scene renderer or physics engine.

Game Object

A game object simply combines a bunch of components. The most common components are going to be the object's transformation, the renderer, physics, and game logic. The game object can also hold a reference back to the scene so the game object can query the scene for information, such as determining if the player is visible to an enemy by accessing the physics engine component of the scene and doing a raycast. The components on a game object should be able to access the other components on the object.

```
class GameObject
{
        String name;
        Component[] components;
        Scene scene = null;

        GameObject()
        {
                // require a game object to have a transform
                this.addComponent(new Transform());
        }

        void addedToScene(Scene newScene)
        {
                scene = newScene;
        }

        void addComponent(component)
        {
                components.add(component);
                component.addedToGameObject(this);
        }
}
```

A transformation holds information about the game object's position, orientation, and size. Other game objects, such the renderer, physics, and game logic will read and write the values stored in the transform.

```
class Transform extends Component
{
        Vector2 position;
        float rotation;
        float size;
}
```

The renderer can be attached to a game object to cause a shape or mesh to be drawn at the location of the transform. A game object's renderer will interact with the scene's renderer component to draw content to the screen.

```
class Renderer extends Component
{
        Mesh mesh;
        Materials[] materials;

        void render(sceneRenderer)
        {
                sceneRenderer.drawMesh(mesh, materials);
        }
}
```

To add game logic, such as moving the player, you simply create a component to control behavior.

```
// class to serve as a base class to all behavoir scripts
class Behavior extends Component
{

}

class PlayerBehavior extends Behavior
{
        void update(float deltaTime, InputState input)
        {
                if (input.keyDown("left"))
                {
                        this.gameObject.tranform.move(-1.0, 0.0);
                }

                // rest of player logic
                // ...
        }
}

class EnemyBehavior extends Behavior
{
        GameObject player;

        void init()
        {
                // searches the scene for an object named player
                player = this.scene.findObject("Player");
        }
```

Now all of the player and enemy logic is contained in their own classes and these parts and components can be easily switched out, modified, and extended.

Conclusion

Writing games is not an easy thing to do, but if you are up to the challenge it can be a very rewarding experience. This article does not tell you everything on how to make a game engine. Most of the classes shown as examples are far from complete. There is still a lot you will need to learn about, such as how to make the different parts of the game engine work together, but hopefully this article has helped give a better picture on how to get started on making a highly resuable core to use on your game projects.

Article Update Log

Jan 11 2014: First published article
Jan 29 2014: Fixed typo with Behavoir class, clarified the intent of the article

License

Comments

Zero_Breaker

Jan 17 2014 01:14 PM

Thank you for this article, it has some very good tips on building game engines. I will add this to my notes while i am creating my own game engine.

jmakitalo

Jan 17 2014 03:02 PM

A nice and brief text about the basics and what not to do. Maybe you could elaborate a bit more on singletons. When to use one and when not. For example, should a console system be a singleton? Pros, cons?

Couldn't agree more on minimizing coupling between different objects.

I hoped to see some more discussion over, e.g., coupling of a scene and other game resources with a player class (which, I guess would here be a derivative of GameObject). Should a player have access to these resources, or just be a dummy container of only player related information? A good example is how to make player attack. This requires that the player is able to check what obstacles and enemies are present. One approach is having player::attack(), where player class would have to have access to a lot of resources. Another approach would be having game::playerAttack(player), where the game class contains the resources and only it will access those. The player class is then just a basic container. Both approaches probably have many pros and cons. I have found the latter easier from the point of view of decoupling, but it's probably bad from the point of view of object oriented design.

slayemin

Jan 17 2014 05:41 PM

This barely scratches the surface of game engines. I think a thorough article on game engines should first ask the question: "What do I want my game engine to do for me?"

Here are a few of my quick answers on game engine requirements:
-Math, Physics and collision detection
-Rendering the scene, lighting, shadows, animation, etc with a high frame rate
-Network code for multiplayer
-Game world management
-GUI management
-Game state management
-Particle effects, sprite rendering, billboards, primitive geometry (points, lines, triangles, quads)
-Importing assets, like sound files, textures, models, font files, shaders, etc. and managing them for me
-Camera interfaces. A camera is a view into the game world. How do I make the camera do what I want it to do?
-IO interfaces for creating log files, saving and loading game states, loading game data files, etc.
-How to use the game engine to create a game (DLL file? stand alone IDE such as Unity? Source code include?). Who is going to be using the game engine? What kind of documentation will they need to use it effectively? What happens if they want to use a feature which isn't supported by the engine?
-How do I create my game world? Do I use an external tool?

I'm sure I'm missing a lot of topics as well. Each of these topics would be an article onto itself, and would be important

to talk about when talking about game engines.

I feel this article is more focused on *software engineering* concepts and *data modelling* as applied to games rather than having much to do with game *engines*. And even at that, it doesn't give me much to go on in terms of figuring out how to architect my games data structures. If that's the direction you're going, a high level overview of the commonly used architectures for games would be very handy. Something along the lines of a hierarchical class model, where we visit each class and talk about how the game uses it, example:
"This is the scene manager. It is responsible for gathering all of the objects in the game world and rendering them onto the screen in accordance with the state of all light sources, camera positions, and game object states. It uses an octree on the backend to efficiently manage the state of the scene, etc. In order for an object to be included on the screen, it *must* be inserted into the scene manager. Should the scene manager be responsible for making sure all objects update themselves? Let's remember that the scene manager is a *scene* manager. Not every object in the game will be a tangible, visible object. So, if you're relying on the scene manager to keep your objects updated, you'll run into a problem with intangible objects requiring an update which don't belong in the scene manager...." etc etc
And, maybe something on game "worlds":
"A game "world" is an enclosed set of game objects which interact with each other in a defined environment with rules. Typically, a single game world is implied for simple games, like pacman, but game world distinctions need to be made when you've got two or more discrete game environments. Take for example, an avatar which moves around on a large map filled with interactable things. When the avatar interacts with an an enemy, a fight begins. We load up the game world for that fight scene and context switch to it. The fight runs its course, and concludes in either a victory or loss. Here, we've got two different game worlds. One can initiate the other. When the large map game world initiates a fight instance, it needs to send some initialization parameters to the fight game world. How do we want to send this data down? When the fight is over, we want the fight game world to send the fight results back up to the map game world (such as items used, health remaining, etc). How do we switch from one game world to another? We want to create a game world "manager" class which is responsible for handling requests to switch between game worlds, loading and unloading assets, loading and unloading game worlds, and helping worlds communicate with each other. Remember as well: Switching to a different *scene* is not the same as switching to another game world. Know the difference."

The conversation about game architecture and class interaction can then transition into a talk about engine capabilities and then into a deep talk about keeping game specific logic separate from engine logic.

Servant of the Lord

Jan 17 2014 05:54 PM

I'm poor when it comes to software architecture/engineering at a higher level, which is why I enjoyed reading your article - thank you for posting it! I found it useful, and +1'd it.

I do have some confusion and questions:

Does bad coupling refer to one class depending on (using) another, or to two classes *mutually* using each other?
Is a Player using a Weapon (via composition) really an example of bad coupling? Player doesn't need internal knowledge of Weapon's implementation details, and the Weapon doesn't know about the player...

I guess my question is, is bad coupling *dependence*, or *inter**dependence*? I always thought it was interdependence, or dependence upon non-interface details.

In your original 'bad' example, the 'Enemy' class was 100% unaware of the player (not coupled in any way).

In your remade 'good' example, the 'EnemyBehavior' inherits 'Behavior' which depends on (but does not own) 'Scene', and the 'EnemyBehavior' assumes/requires that certain data exists in 'Scene'. Instead of being coupled to compile-time

code, it's now coupled to run-time data: It assumes that "Player" is a string-key that exists within the shared Scene that it inherits but does not own.

As an example of de-coupling, is that actually better? Why is it better?

I'm not arguing against component-based design and data-driven design - but I don't want my *bugs* to be data driven - that makes them much harder to track down. =)
I want my errors to be caught compile-time as much as possible (though they can't be 100% compile-time).

[Edit:] Upon re-reading your article, I think I mixed up your cohesion and coupling sections in my mind, and was comparing your 'bad' cohesion to your final 'Game Engine Design' example.

It seems like your article is talking about two seperate things:
1) General-purpose guidelines for code quality.
2) Possible structures for game engines, using component-entity-systems.

I seem to have been mixing up the examples between the two sections! Whoops. Maybe they should each be separated articles so you could go deeper into each topic separately?

HappyCoder

Jan 18 2014 06:14 PM

@slayemin

I realize that this is not a comprehensive game engine making tutorial. It is meant to only explain a certain aspect of a game engine. I do plan on making more articles detailing the different parts.

As for the intangible objects, under the design I described, an game object with only a behavior script is valid. You don't need to attach a rendering or physics component to any given game object

@Servant of the Lord
I guess you do bring up a good point on dependance vs interdependance. I definitely agree that interdependance is a problem. It can quickly make a program a nightmare to manage. It is sometimes necessary. At those times, I try to make some sort of interface to send messages to parent objects. That way the class only talks to a generic interface and doesn't directly depend on concrete class.

I think I will adjust the article based on that feedback, thanks.

Rohithzhere

Jan 20 2014 12:55 AM

Thanks for this neat article!! More articles like these would definitely be of great help!!

blue_charmander

Jan 20 2014 06:03 AM

Wonderful article, it is refreshing to see an article that isnt discouraging people to create their own game engines. Thanks

h0wser

Jan 20 2014 04:30 PM

Do you have anything to say about using an event architecture in games? Both events that support a callback so that each event triggers a list of functions to execute, and the type where you poll for events. What are the does and don'ts when it comes to events?

Nathan2222_old

Jan 22 2014 10:25 AM

> **slayemin, on 18 Jan 2014 - 07:41 AM, said:**
>
> This barely scratches the surface of game engines. I think a thorough article on game engines should first ask the question: "What do I want my game engine to do for me?"
> Here are a few of my quick answers on game engine requirements:
> -Math, Physics and collision detection
> -Rendering the scene, lighting, shadows, animation, etc with a high frame rate
> -Network code for multiplayer
> -Game world management
> -GUI management
> -Game state management
> -Particle effects, sprite rendering, billboards, primitive geometry (points, lines, triangles, quads)
> -Importing assets, like sound files, textures, models, font files, shaders, etc. and managing them for me
> -Camera interfaces. A camera is a view into the game world. How do I make the camera do what I want it to do?
> -IO interfaces for creating log files, saving and loading game states, loading game data files, etc.
> -How to use the game engine to create a game (DLL file? stand alone IDE such as Unity? Source code include?). Who is going to be using the game engine? What kind of documentation will they need to use it effectively? What happens if they want to use a feature which isn't supported by the engine?
> -How do I create my game world? Do I use an external tool?
> I'm sure I'm missing a lot of topics as well. Each of these topics would be an article onto itself, and would be important to talk about when talking about game engines.
> I feel this article is more focused on *software engineering* concepts and *data modelling* as applied to games rather than having much to do with game *engines*. And even at that, it doesn't give me much to go on in terms of figuring out how to architect my games data structures. If that's the direction you're going, a high level overview of the commonly used architectures for games would be very handy. Something along the lines of a hierarchical class model, where we visit each class and talk about how the game uses it, example:
> "This is the scene manager. It is responsible for gathering all of the objects in the game world and rendering them onto the screen in accordance with the state of all light sources, camera positions, and game object states. It uses an octree on the backend to efficiently manage the state of the scene, etc. In order for an object to be included on the screen, it *must* be inserted into the scene manager. Should the scene manager be responsible for making sure all objects update themselves? Let's remember that the scene manager is a *scene* manager. Not every object in the game will be a tangible, visible object. So, if you're relying on the scene manager to keep your objects updated, you'll run into a problem with intangible objects requiring an update which don't belong in the scene manager...." etc etc
> And, maybe something on game "worlds":
> "A game "world" is an enclosed set of game objects which interact with each other in a defined environment with rules. Typically, a single game world is implied for simple games, like pacman, but game world distinctions need to be made when you've got two or more discrete game environments. Take for example, an avatar which moves around on a large map filled

> with interactable things. When the avatar interacts with an an enemy, a fight begins. We load up the game world for that fight scene and context switch to it. The fight runs its course, and concludes in either a victory or loss. Here, we've got two different game worlds. One can initiate the other. When the large map game world initiates a fight instance, it needs to send some initialization parameters to the fight game world. How do we want to send this data down? When the fight is over, we want the fight game world to send the fight results back up to the map game world (such as items used, health remaining, etc). How do we switch from one game world to another? We want to create a game world "manager" class which is responsible for handling requests to switch between game worlds, loading and unloading assets, loading and unloading game worlds, and helping worlds communicate with each other. Remember as well: Switching to a different *scene* is not the same as switching to another game world. Know the difference."
> The conversation about game architecture and class interaction can then transition into a talk about engine capabilities and then into a deep talk about keeping game specific logic separate from engine logic.

Better

Keyeszx

Jan 22 2014 05:18 PM

Can someone explain what the scene is? I don't quite understand how everything is supposed to connect together. Like how do things get added to the scene? It looks like the scene itself gets told to call a method in the gameobject class to add that object to the scene. But what tells the scene which gameobject to add?

Michael Tanczos

Jan 27 2014 08:30 AM

Perhaps the expectations are too high for an article of this type. The moderation right now seems a bit heavy-handed IMO.. I'd like to see where subsequent articles go rather than to make this into a monster first article. I think there is more to gain from having it around than relegating it to the edit bin.

Dave Hunt

Jan 28 2014 03:18 PM

Fix this and I'll be glad to approve the article:

```
// class to serve as a base class to all behavoir scripts
class PlayerBehavior extends Behavior
{

}
```

;-)

d4n1

Feb 03 2014 09:17 AM

In your section about Coupling, I think you should give a mention about the difference between 'tightly coupled' and 'loosely coupled' objects. The way that you are saying it makes it sound like the programmer should never reference an object, however I think what you should be saying to the reader is that a programmer should try to abstractly reference (or indirectly reference) an object as much as possible. And to do this, you should learn to program through interfaces (and of course give an example of something).

reigota

Feb 04 2014 04:26 AM

Impressive how these objects looks like Unity3D's object!

Good article, I liked it!

Irlan Robson

Mar 20 2014 03:28 PM

I've used a similar approach but instead of passing a Scene* to the Behaviour class I passed a Pathfinder* (containing the goal node, alternative nodes position of the path). That way the enemies know what target to follow and doesn't know about the player just about a target (a 3D vector).

You could pass a Scene* pointer to the Pathfinder class too. I think this sounds more elegant. :-)

Madolite

Sep 20 2014 10:21 AM

I was just about to ask a question about this stuff, but then I saw this article, so this is nice.

Note: Please offer only **positive**, **constructive** comments - we are looking to promote a positive atmosphere where collaboration is valued above all else.