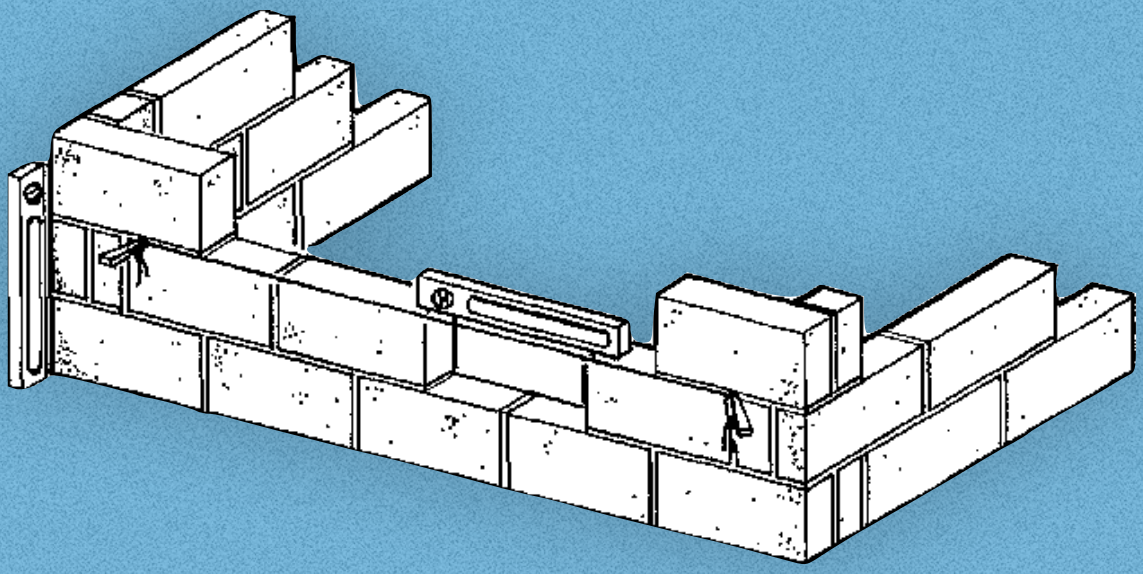
LUDUM ENGINE

ARCHITECTURE & USAGE DESCRIPTION



BUILD A GAME IN A DAY

VIKTOR LYRESTEN JACOB NILSSON ANTON BERG JOSEF HOLMÉR

- Boom shackalack, ja tack!

Changes in version two:

Added a new section in Architectural Overview called 'Concrete componenets included in the engine'.

Added an example as to how scenes can be seen as an implementation of the 'state' design pattern, in the section 'Scenes' of Architectural Overview.

Added use cases for components related to Movement(including Concrete player triggered action), Collisions, Sound and Health.

The Update Use Case has been updated with more detail.

The Draw Use Case has been updated with more detail.

The Class Diagram has been slightly updated.

The Interaction diagrams have been updated to reflect changes in the use cases.

Contents

Introduction

'Ludum' is the latin word for a game and playing, so a ludum engine is a motor for driving a game! Specifically, it's an engine for making any form of 2d pc game in C #, and making them fast!

While reading this document, at least basic knowledge of the C # programming language and knowledge of object oriented design and design patterns are expected.

The engine extends the Mono Game framework and aim for the following result:

1. Fast development time.
2. Support for all major PC platforms.
3. Easier collaboration between programmers and game designers.
4. Game extendability.

This document gives an overview of the architecture and usage of the engine and motivates the design choices made, mostly in relation to these goals.

I Fast development time

Obviously using a framework should lead to less work for the developer. The Ludum Engine further cuts development time from Mono Game by a standardized usage and focus on 2d development.

II Support for all mayor PC platforms.

This means support for Windows, Mac OS X & Linux operating systems and their standard input and output devices (keyboard, mouse, windowing system). MonoGame also supports other platforms, these are not supported when using the Ludum Engine.

III Easier collaboration between programmers and game designers.

The Ludum engine architecture is designed so that creating a new game uses an easily understood declarative format, that requires little 'actual' programming to be understood by designers.

IV Game Extendability

It should be easy to extend a game as needed, or make a new game with the same resources and code base. This is accomplished thanks to the modular nature of the Ludum engines architecture.

Architectural Overview

The Ludum Engine is built on the philosophy of splitting distinct functionality of game objects into components that can be reused and linked together to form different entities such as missiles, tanks, jetpacks, cats & dogs. The core idea of component based design is perhaps best described by Mick West in *"Evolve Your Hierarchy"* (West 2007).

Entities and Entity Blueprints

The Ludum engine implements a class called Entity Blueprint that is a combination of the *composite* design pattern, the *factory* pattern and the *prototype* pattern. It is created from the engines BlueprintManager.

The blueprint allows instances of any type of component to be created and kept in a ComponentCollection, that in practice is an extended dictionary for components. These components are instantiated, but never initialized while in the collection of the blueprint. Being instantiated, the properties of a component can be added independently for different blueprints programmatically. This was a design goal of the engine to make a game more quickly written and readable and thus support goal 1 & 4.

Entity is a sealed class to avoid it from becoming anything other than a pure aggregation of components. It contains a ComponentCollection that is copied from an EntityBlueprint, during which all components also becomes initialized. Because Blueprints acts as prototypes for actual entities, it is from a blueprint an entity is created. An entity is also a combination of the *composite* & *factory* patterns but does not act as prototypes like blueprints do.

The reason for storing components in classes like ComponentCollection, Entity and EntityBlueprint instead of storing all components in a manager and accessing them with id numbers (West 2007 & Wenderlich 2013) is for increased readability (goal 3) and also easier debugging in the .net runtime and stronger typing, witch leads to fewer errors being made during development and therefore shortens the development time (goal 1). This is supported in a study by Kleinschmager et al. but also because the prototype pattern allows for easy dependency control between components as recommended by Passos et al. (2009)

Because the components are added to a blueprint before they are initialized, the dependency check does not have any performance impact (although the components can also be added and removed on the fly)

Components

In the Ludum engine a Component is an abstract and extendable class. A few properties, fields and methods are defined in the base class itself that are needed for the engine to function; this includes a reference to the entity the component belongs to if initialized, methods for cloning and deleting the component. A list of all event subscriptions and a method for adding new ones. It also includes a list of all components that are dependent on the instance.

Because the Component class is meant to be extended into a wide array of different components with various functionality, it contains an abstract method called Initialize that needs to be overridden, it will be called when a component is copied from an EntityBlueprint to an Entity and becomes initialized. The Initialize method replaces the usage of a constructor.

Many times a component will need other components in the same entity to function correctly. Passos et al. (2009) suggests the usage of the *dependency injection* pattern to make sure that the game does not break when a required component does not exist. This pattern is implemented in the Components base class and a protected method can be called from a components subclass to register itself as a client dependent on another components service. If the service component ceases to exist the client component is also deleted.

The Component class is an example of the *template* design pattern and to some extent the *visitor* pattern, and it includes an implementation of the *dependency injection* pattern.

The Ludum Engine comes with a few standard components, these are: Sound, Music, Sprite, Position, Size and CollisionRectangle. How to extend the component class is described in the usage section of this document. The included components are described further ahead in this chapter.

Communication and Events

Components can communicate with each other by registering itself as dependent on another components services, this way the public properties and methods of the service component becomes available directly to the client component. This works well in situations where the client component is interested in accessing the service on set intervals, but for situations where the client either needs to monitor the state of another component or respond to a one time occurrence its better to implement the *observer* pattern.

The Ludum engine includes an Event class; a component can subscribe to an event of another component by assigning one of its methods to the event. When the event is

triggered from the publishing component the method of the subscribing component is executed. This allows for a very flexible way of extending functionality from a component to others; as an example, a Health component attached to a monster entity could create an OnDeath event that is called when its value reaches zero, a Loot component could also be assigned to the monster that waits until it has died and gives the player some kind of reward on the OnDeath event.

Because event focused architectures are sensitive to the lapsed listener problem, all game objects (Scenes, Entities, EntityBlueprints, Components, ComponentCollections and Events) include a delete method. This makes sense for a game as you often want to explicitly dispose of objects and resources.

Scenes

In the Ludum engine Scenes are concrete sealed classes that are created from the SceneManager. A Scene represents the current screen, space, room or level, that the game is currently at. When entities are instantiated from blueprints they are tied to one specific scene and only one scene is active at a time. A scene contains events that can be subscribed to by its components, these are: OnUpdate, OnDraw, OnLoad and OnUnload.

The SceneManager controls what scene is active and can be used to go from one scene to another. When going to a new scene, the OnUnload event is first triggered for the current scene, followed by the OnLoad event for the new, now active scene.

As long as the scene is active its OnDraw and OnUpdate events are triggered continuously. But as soon as it becomes inactive all its components are effectively paused. The SceneManager can also ignore the previous scenes unload method so that switching back to a previous scene can be done almost seamlessly without loading times.
Scenes can be used to implement anything between game stages to pause screens. They are very useful and can be seen as an implementation of the *state* pattern for the game itself.

Imagine a pause menu with the choices of continuing the game or quiting, it can be implemented by creating a new scene with two entities, one for each choice. The entities would include components for drawing text and registring mouse clicks, each entity would go to another scene when pressed, one to the main menu scene and one back to the previous scene.
In the game a player entity would have a pause component that would simply move to pause menu scene when the escape button is pressed, the scene containing the player character would be paused as the pause menu scene is entered and resumed when the continue entity is pressed.

Systems

When designing the engine the approach of keeping data and functionality in the components was chosen over the entity / system approach described by Ray Wenderlich (Wenderlich 2013). This was due to concerns over extendability (goal 4) and the overall flexibility of the game engine, if all actual programing was done in systems these systems would need to be extended and written differently for each game, which seemed to limit the reusability of the code written. By keeping all code that affects the behavior of the entity in components, it can be reused many times in many games.

Ludum engine still implements several systems, or managers. These perform operations that affect the entire game space. For example switching the current scene, loading and unloading content

The Settings Manager is a static class that has public fields for controlling the game engines starting options, it is accessed by pretty much all other managers during the initialization process.

The Game Manager is the link between Mono Game and the ludum engine, it keeps track of the game time and initializes the resource and render managers during startup. It triggers the update loop of the current scene.

The Render Manager controls global drawing options, handles the game window and resolution. It triggers the draw loop of the current scene.

The Audio Manager controls the global volume and can be used to pause/resume/stop all existing Sound and/or Music components.

The Error Handler defines a common way for other parts of the game engine to throw errors and outputs them in a standardized way. The verbose level can also be set here.

The Input Handler keeps track of the mouse positions and runs events that components can subscribe to for all available button and keyboard presses.

The Blueprint Manager creates, copies, deletes and stores EntityBlueprints.

The Scene Manager creates and deletes scenes and keeps track of the current scene. It triggers the current scenes load and unload methods.

Finally the Resource Manager makes methods available to components for loading, unloading, storing and retrieving file resources such as sounds and 2d textures. It makes sure the same resource is not loaded multiple times.
There is also a static class called Ludum, which acts as a *facade* for all the systems and keeps an instance of each one to make them available globally. It contains two methods, one for initializing the engine and one for actually starting the game.

Concrete componenets included in the engine

The Ludum Engine comes with a few built in and ready to use components, here they are briefly described.

Sound
Allows an entity to play sounds at demand, it specifies a sound file to load and contains methods for playing/stoping/pausing the sound and for adjusting its volume. It listens to global events in the audiomanager which allows the audiomanager to play/pause/stop all sounds at a global level (the audio manager also contains similar events for music components and for controlling both sound & music components).

Music
Works exactly like the Sound component, the first of differences is that it listens to the music events in the audioManager instead of the sound events. The second difference is that the music component can be looped by setting a property.

Position
Declares an entitys position in 2d space, it contains X and Y coordinates as properties. It is required for both the Sprite and the CollisionRectangle components.

Size
The size components describes the width and height of a entity, it contains both as float properties, it is required for having a collision rectangle.

CollisionRectangle
The collision rectangle allows events to be triggered when the entity collides with another entity. It contains a OnCollision event that triggers during a collision and a property that contains the colliding entity. Another component can listen to the OnCollision event and during it triggering check the CollidingEntity property of the CollisionRectangle to determine what kind of collision is occuring. The Rectangle can be set up to check for certain collisions automaticly, for example the player can check for collisions with enemies. Collisions can also be checked manually with the components Intersects method.

Sprite
The sprite allows the entity to be represented on the screen with an image texture. It requires that the entity has a position and will create one if it does not exist (at x: 0 & y: 0). It specifies a filename to load a texture from and a depth that controlls how close to the camera it will be drawn (sprites with lower dephts will drawn over sprites with higher depths). The sprite subscribes to the draw event of a scene and adds the texture to the rendermanagers spritebatch during the draw loop.

Using the Game Engine

This sections is intended as a simple guide to getting up an running with the ludum engine from a usage point of view, for a more technical explanation of how the engine actually works during these procedures please refer to the included Use Cases in Further Insights.

Creating your project

* Create a new empty project in either Mono develop, Xamarin Studio or Visual Studio, make sure your environment supports .net version 4.0 and that you have the latest version of Mono and Mono Game installed.
* Import the LudumEngine assembly into your project.
* Done, you are free to structure your application in any way you want. This guide just assumes that the code is inserted somewhere at the same place and accessed through main.

Changing settings

The first thing you want to do is configure the game settings. this is done by accessing Ludum.Settings and changing its various properties, and when you are happy with your settings, initialize the engine with Ludum.Initialize();

```
Ludum.Settings.Fullscreen = true;
Ludum.Settings.AssetsDirectory = "GameContents";
Ludum.Initialize();
```

Creating blueprints and adding components

Your game needs some blueprints to be able to spawn game objects, blueprints are created by calling Ludum.Blueprints as following:

```
EntityBlueprint Tyrannosaurus = Ludum.Blueprints.CreateEntityBlueprint ("TRex");

EntityBlueprint GreatApe = Ludum.Blueprints.CreateEntityBlueprint ("GreatApe");

Tyrannosaurus.AddComponent<Sprite> ().Set(filename: "trexSprite");
Tyrannosaurus.AddComponent<Position> ().Set(x: 0, y: 50);
Tyrannosaurus.AddComponent<Health> ().Set(startingValue: 100);
Tyrannosaurus.AddComponent<AttackEverything> ();

GreatApe.AddComponent<Sprite> ().Set(filename: "apaSprite");
GreatApe.AddComponent<Position> ().Set(x: 200, y: 50);
GreatApe.AddComponent<Health> ().Set(startingValue: 200);
GreatApe.AddComponent<AttackEverything> ();
```

What happens here is that two Blueprints are created, one stored as "TRex" and the other as "GreatApe", we also assign them to variables so that we easily can add components to them.

We give the Tyrannosaurus 4 components, a position, a sprite for drawing, (which we will load from the file trexSprite in our content folder). A health component, and a component that will make it attack everything.

The GreatApe blueprint is given the same components, but we load a different sprite and set the starting health to 200 instead of 100 as for the trex.

Creating scenes

Next, we need a scene for the game to run. We are only going to add one scene where the fight will take place, this is done at Ludum.Scenes:

```
Scene arena = Ludum.Scenes.CreateScene("arena");
```

Creating entities

The scene is going to look pretty empty without any objects in it, so we create instances from our already completed blueprints:

```
Entity KingKong = GreatApe.Instantiate(scene: arena);
Entity Trex = Tyrannosaurus.Instantiate(scene: arena);

Random r = new Random();

if (r.Next(0,1) > 0) {
       Trex.CreateComponent<LaserWeapons>();
}
```

This adds a GreatApe and a Tyrannosaurus to the arena scene, with a 50 percent chance of this specific Trex being equipped with some much needed laser weapons.

Running the game

```
Ludum.Start(scene: arena);
```

This starts the game with the arena as the first loaded scene. Ta-Da!

Creating new components

To create a new component, first create a new class in your own namespace and make sure its using LudumEngine. Make sure the class is extending the base Component class and overrides the Initialize() method.

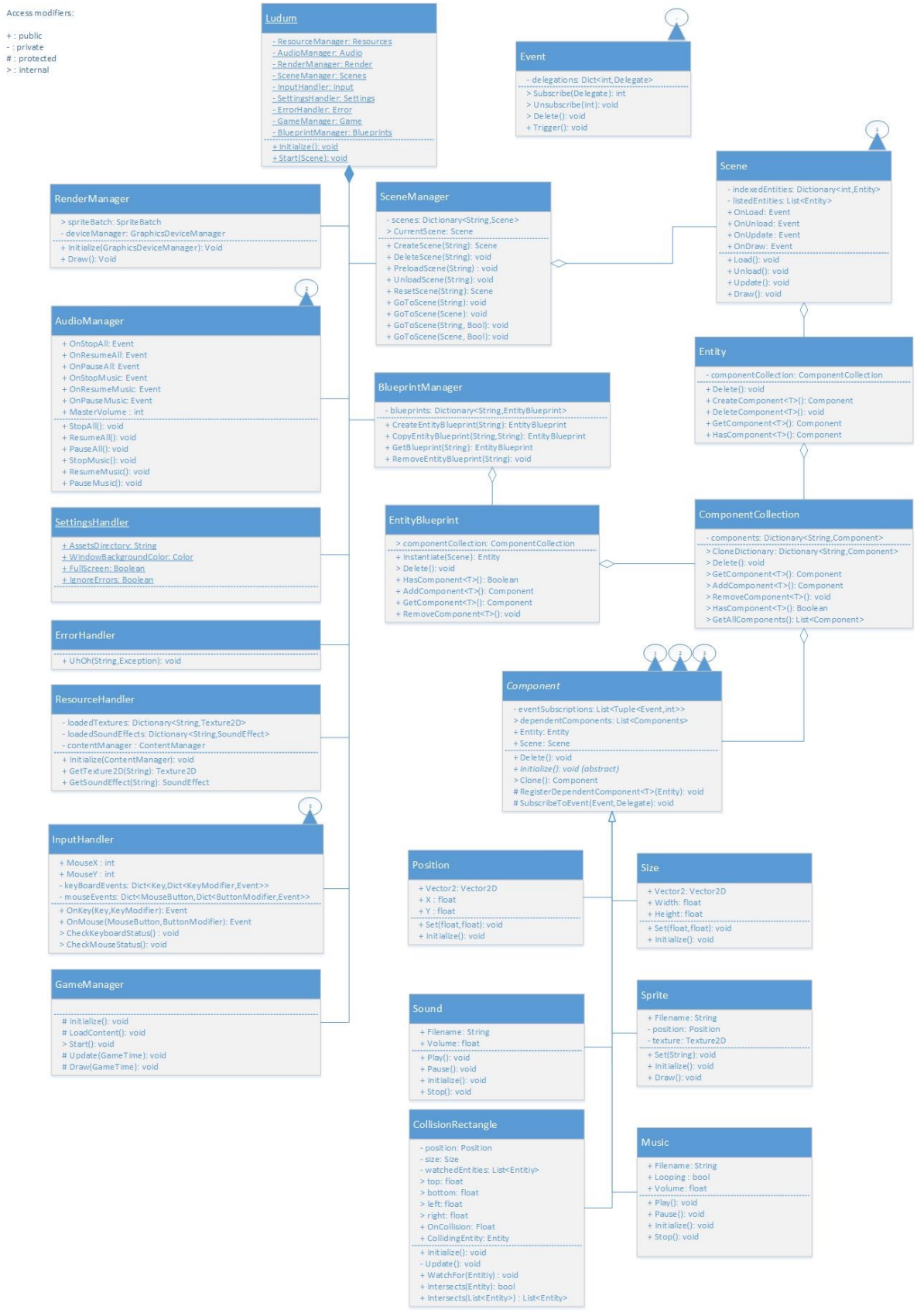
```
using System;
using LudumEngine;

namespace CosmicEncounter {
	public class Health : Component {
		public sealed override void Initialize() {}
	}
}
```

Thats it! Your component can define any method or property it needs, if your property implements a lot of properties it is recommended that you create a Set method as an alternative to a constructor.

```
public class Health : Component
	{
		public int StartingValue { get; set; }
		public Event OnDeath;

		private int _value;

		public void Set(int startingValue = 100) {
			this.StartingValue = startingValue;
		}

		public sealed override void Initialize() {
			OnDeath = new Event ();
			Value = StartingValue;
		}

		public bool IsDead { get; private set;} = false;

		public int Value {

			get { return _value; }

			set {
				_value = value;

				if (_value <= 0) {

					//Trigger death if the health value passes below 0
					if (!IsDead)
						OnDeath.Trigger ();

					IsDead = true;
				}  else {
					IsDead = false;
				}
			}
		}
	}
```

Further Insights

Class Diagram



Use Cases

Martin Flower style notation.

Start game

Main Success Scenario:

1. Ludum Engine is initialized

a. LudumEngine constructs a SettingsManager object and desired settings is set.

b. LudumEngine constructs a GameManager object.

c. LudumEngine constructs a ResourceManager object which sets the desired content directory from the LudumEngines SettingsManager.

d. LudumEngine constructs a RenderManager object which sets events and graphics attributes based on the LudumEngines SettingsManager.

e. LudumEngine constructs a AudioManager object.

f. LudumEngine constructs a InputManager object which initializes dictionaries to keyboard buttons and mouse events containing events as values.

g. LudumEngine constructs a EntityFactory object.

h. LudumEngine constructs a SceneManager.

i. LudumEngine constructs a BlueprintManager

2. LudumEngine is started with a game

a. LudumEngines GameManager is started and the GameManager calls its Game superclass' run method to start the game.

b. The GameManagers Initialize method is called

i. Its superclass is initialized

ii. The RenderManager is initialized

iii. The ResourceManager is initialized and its ContentManager with directory is set.

iv. The BlueprintManager is initialized and blueprints are created

v. The SceneManager is initialized and scenes with entities are added

3. The GameManagers LoadContent method is called

a. The superclass' content is loaded.

4. The update loop runs

a. The GameManagers Update method is called.

b. The GameManagers Draw method is called.

Update

Main Success Scenario:

All components in the current scene that are supposed to get updated gets updated.

1. **GameManager calls the CheckKeyboardStatus and CheckMouseStatus methods of the InputManager**

2. **InputManager checks the keyboard status and the status of each possible key**
   a. If the key has been released, pressed or is being held, the corresponding event is triggered
      i. All subscribed components run their method/s associated with the triggered event

3. **InputManager checks the mouse status and updates the mouse x and y positions**

4. **InputManager checks the status of each possible mouse button**
   a. If the button has been released, pressed or is being held, trigger the corresponding event
      i. All subscribed components run their method/s associated with the triggered event

5. **GameManager calls the Update method of the current scene in the SceneManager**

6. **If the scene is not paused it triggers its OnDraw event**
   a. All subscribed components run their method/s associated with the triggered event

Draw

Main Success Scenario:

All textures in the current scene which are supposed to be drawn are drawn.

1. **GameManager calls the Draw method in the RenderManager**

2. **RenderManager clears the screen and draws the specified background color for the window**

3. **RenderManager begins the spritebatch drawing using the correct sortmode and aplha blending**

4. **RenderManager calls the Draw method of the current scene in the SceneManager**

5. **If the scene is not paused it triggers its OnDraw event**

6. **All sprite components that subscribe to the event run their drawing methods**

7. **Each sprite adds a texture2d at a position, size and depth to the spritebatch**

8. **RenderManager ends the spriteBatch's drawing**
   a) The SpriteBatch sorts all textures according to depth
   b) All added textures are drawn in the correct order

Note: LudumEngine draws all, even occluded, textures. However, the XNA graphics device make sure that textures outside the viewport won't be drawn.

Movement *Assumes the creation of a movement component!*

Initialize Move component

Main Success Scenario:

1. **The Move Component's velocity is set to the velocity input parameter**
2. **The Move Component requests its entity's Position Component from its Entity**
3. **The Move Component creates a previousPosition variable and sets it to its Entity's position**
4. **The Move Component subscribes to its scene's update event with its update function**
5. **The Move Component subscribes to a set of key's pressed event in Ludum's InputHandler with respective move functions for each movable direction**

Update Position component

Main Success Scenario:

1. **The Move Component's Scene's update event triggers**
2. **The Move Component sets its previousPosition variable to its entity's position**
3. **The Move Component update its entity's position with the Move Component's velocity**

Movement - Change Velocity's Direction

Main Success Scenario:

1. **One of the key's (the Move Component subscribed to) pressed event is triggered**
2. **The Move Component changes the velocity's direction by changing its X and Y variables so that the desired direction is achieved**

Collisions

Initialize Collision Component

Main Success Scenario:

1. **A new onCollision event is constructed**
2. **The position component from the entity holding the ColisionRectangle component are set as the position field in the component using the GetNeededComponent**
3. **The size component from the entity holding the ColisionRectangle component are set as the size field in the component using the GetNeededComponent method**
4. **The update method in the CollisionRectangle is subscrided to the update event of the Scene in which the component belongs to**
5. **The other entities which shall be controlled for collisions with, are added to the watchedEntities list**

Controll Collisions

Main Success Scenario:

1. **The CollisionRectangles update method is invoked**
2. **The entities in the watchedEntities list are controlled for collisions by controlling if the other entities CollisionRectangles intersects with this entities CollisionRectangle.**
3. **The CollisionRectangle components OnCollision event is triggered.**

Health *Assumes the creation of a health component!*

Initialize Health Component

Main Success Scenario:

1. **The Health Component's amount variable is set to the health input parameter**
2. **The Health Component requests its entity's CollisionRectangle which handles collision with dangerous entities**
3. **The Health Component subscribes to the aforementioned CollisionRectangle component's OnCollision event with its onDamage function**

Update Healthcomponent

Main Success Scenario:

1. **The CollisionRectangle component's (which handles collition with dangerous entitys) onCollition event is triggered**
2. **The CollisionRectangle component requests all colliding entities from the CollisionRectangle component**
3. **For each colliding Entity, The CollisionRectangle component requests its DamageComponent**

4. **For each DamageComponent, The CollisionRectangle component removes the specified damage amount from the HealthComponent's amount variable**

Sound

Initialize Sound effect

Main Success Scenario:

1. **The components soundeffect variable is set by calling the ResourceManager's GetSoundEffect method, where the sound file is loaded**
2. **The sound effects volume is set to the volume input parameter**
3. **The sound components pause method is subscrided to the AudioManagers OnPauseSounds Event variable**
4. **The sound components play method is subscrided to the AudioManagers OnPlaySounds Event variable**
5. **The sound components stop method is subscrided to the AudioManagers OnStopSounds Event variable**

Play movement sound effect when moving

Main Success Scenario:

1. **The Movement Components MoveRight method is invoked**
2. **The Movement Components MoveRight method calls on the GetNeededComponent method to access the Movement Sound Component**
3. **The Movement Sound Componens Play method is called**
4. **The sound effect is played.**

Create EntityBlueprint

Main Success Scenario:

1. **A BluePrint is created**
2. **A ComponentCollection for the Blueprint is created**
3. **The BluePrint are added to the BlueprintManagers blueprint collection**
4. **The Components, which the user has decided shall belong to the Blueprint, is created**
    a. The Components are added to the ComponentCollection
    b. The Components values are set

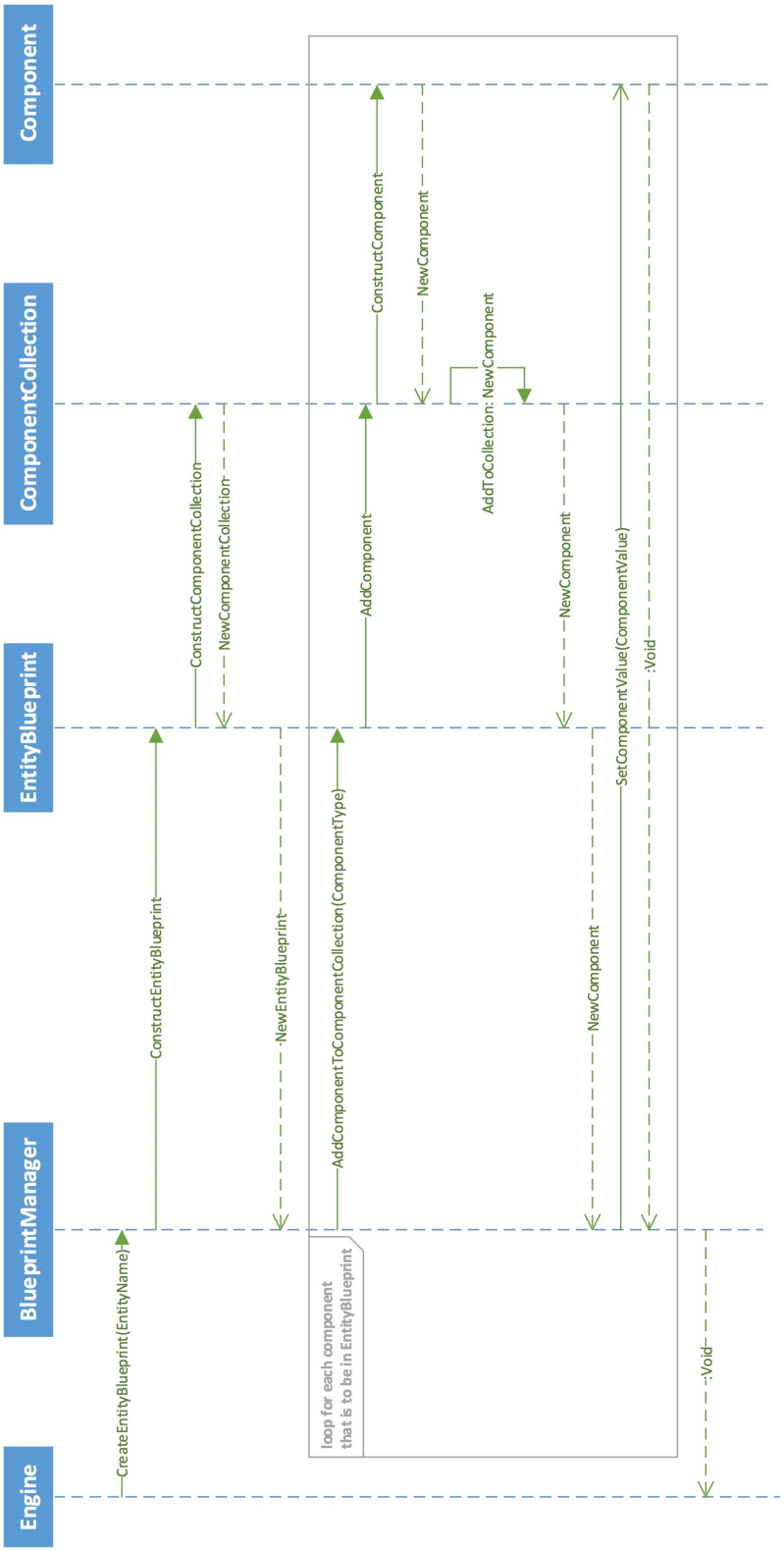
Load entity to a scene

Main Success Scenario:

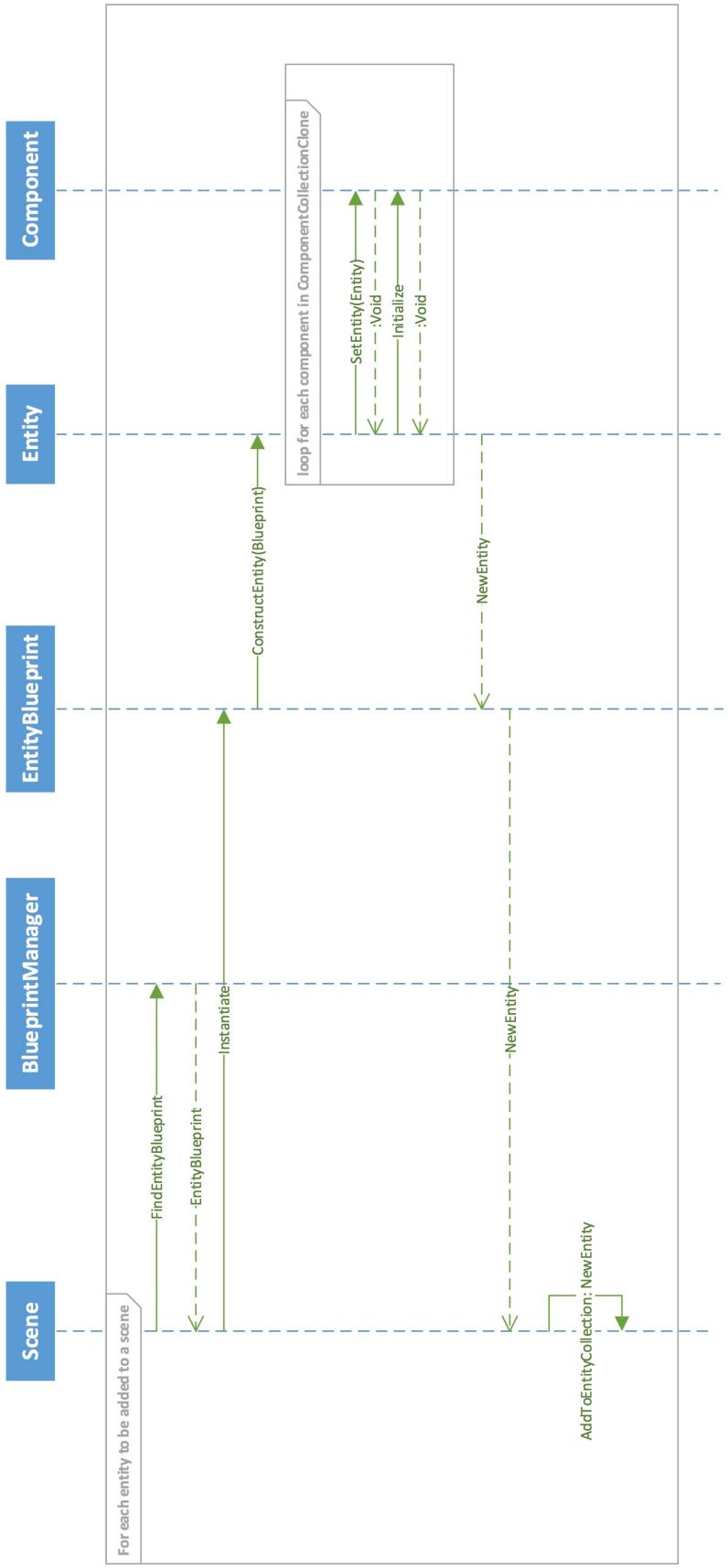
1. **An Entity is created by looking up and instantiating the desired EntityBlueprint from the BlueprintManagers BlueprintCollection.**
    a. The blueprints ComponentCollection is cloned and set as the Entities ComponentCollection.
    b. The components in the Entities ComponentsCollect gets their EntityProperty set as the entity holding it.
    c. All components are initialized.
2. **The entity is added to the the Scenes EntityCollection.**

Interaction Diagrams

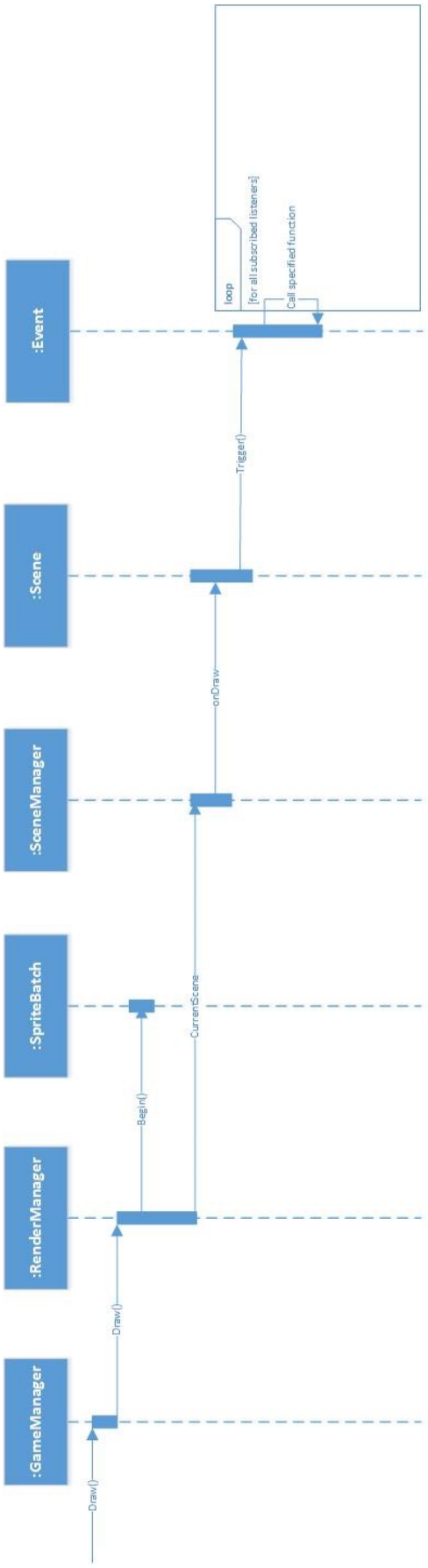
CreateEntityBlueprint

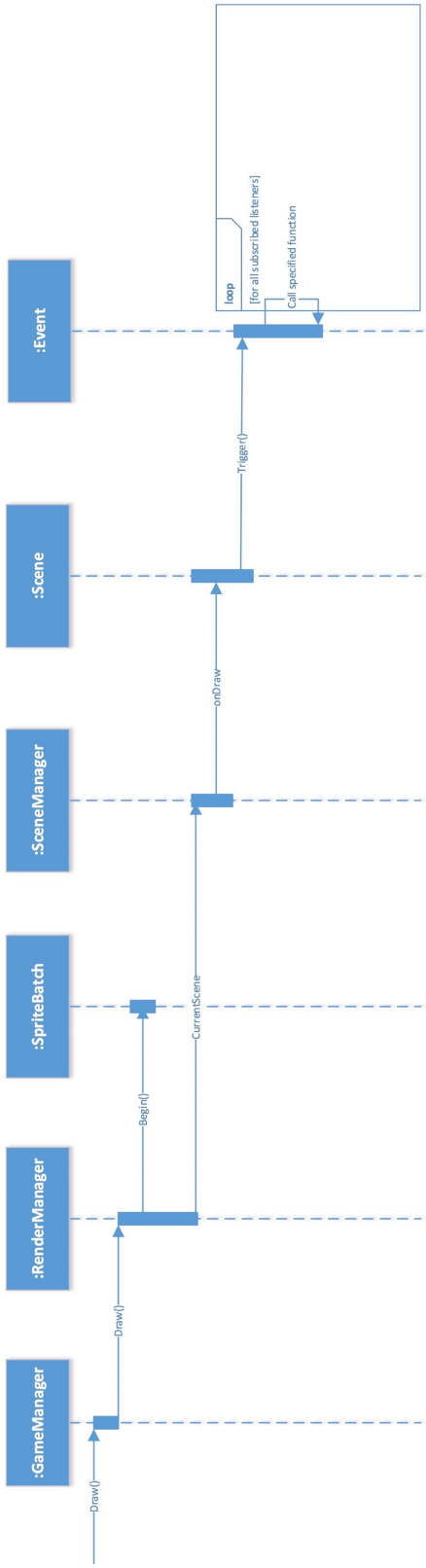


LoadEntitysToScene

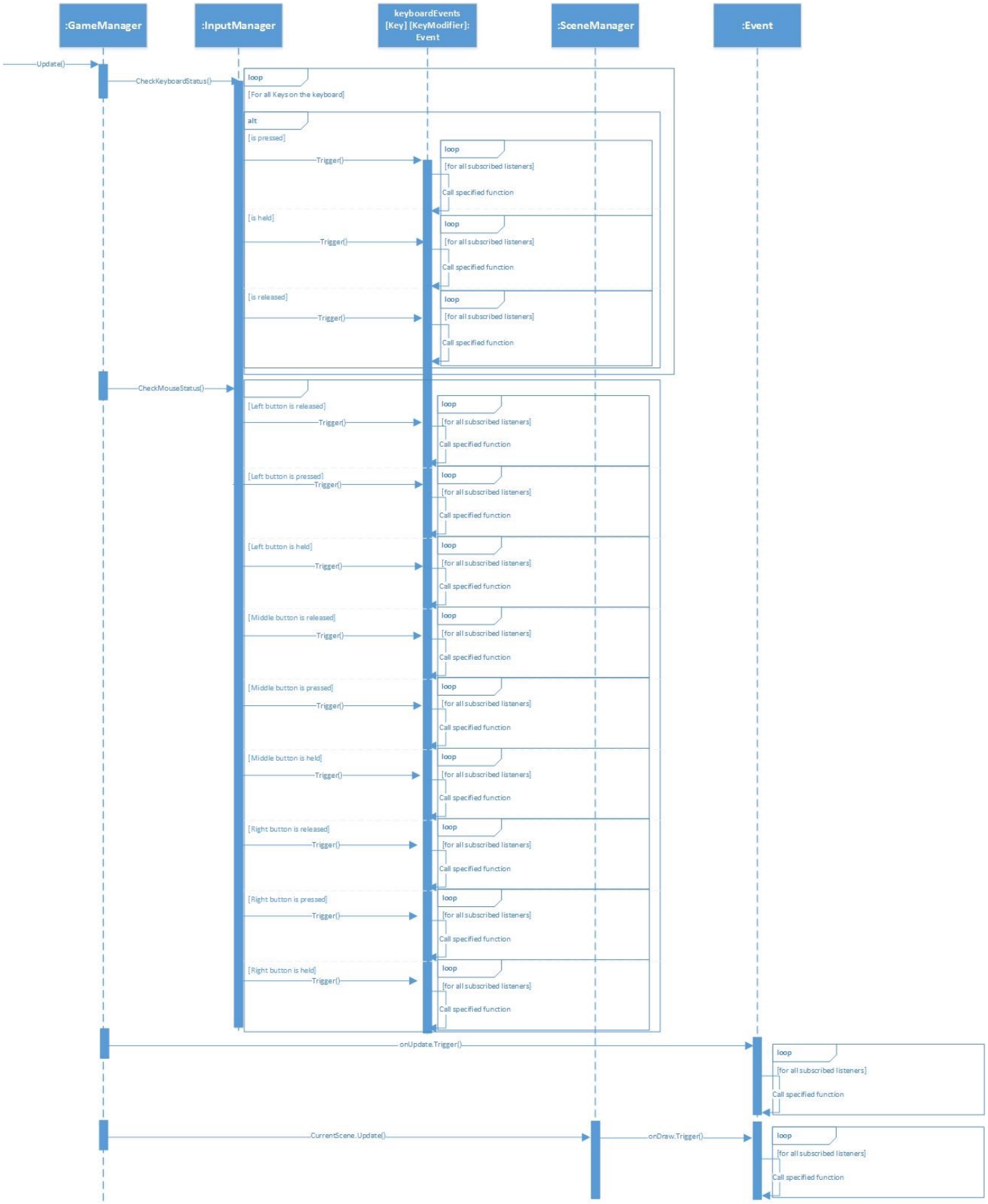


Draw


loop
[for all subscribed listeners]
Call specified function
:Event
Trigger()
:Scene
onDraw
:SceneManager
CurrentScene
:SpriteBatch
Begin()
:RenderManager
Draw()
:GameManager
Draw()


:GameManager
:RenderManager
:SpriteBatch
:SceneManager
:Scene
:Event
Draw()
Draw()
Begin()
CurrentScene
onDraw
Trigger()
loop
[for all subscribed listeners]
Call specified function

Update


:GameManager
:InputManager
keyboardEvents [Key] [KeyModifier]: Event
:SceneManager
:Event
Update()
CheckKeyboardStatus()
loop
[For all Keys on the keyboard]
alt
[is pressed]
Trigger()
loop
[for all subscribed listeners]
Call specified function
[is held]
[is released]
CheckMouseStatus()
[Left button is released]
[Left button is pressed]
[Left button is held]
[Middle button is released]
[Middle button is pressed]
[Middle button is held]
[Right button is released]
[Right button is pressed]
[Right button is held]
onUpdate.Trigger()
CurrentScene.Update()
onDraw.Trigger()

Changes from assignments specifications

The assignment specified a minimum number of systems that should exist in the system and their functionality. This section lists those systems and changes made from the requested functionality.

Render Manager

The RenderManager does not initialize 2d graphical resources, this is handled by the resource manager.
Position, transparency and other graphical effects are handled directly in the affected components.

Input Manager

The input manager does not translate input as abstractly as to "shoot" or to "walk". But it creates events for each available keyboard and mouse buttons that can be triggered (either on press, release or hold), to make sure the system works consistently.
While the engine only target personal computer platforms, it can easily be extend to support alternative input methods and link the new inputs to the already existing events.

Sound Manager

The Sound Manager is called the Audio Manager. Each component that loads and plays some sort of audio does this independently and uses the resource manager and not the Audio Manager for this. But the Audio Manager can be used to stop and resume all audio (or just all music) at once and controls the master volume of the program.

Scene Manager

Positions are handled with the position component, so there are no graphs in scenes or layers. Layers does not even exist, depth is handled in the components by the use of z positioning. Time is managed by the GameManager, and XNA.

Physics Manager

The physics manager has been removed. As the engine implements functionality directly in components it seems unnecessary, instead a component named CollisionRectangle exists that handles collision, and then more components can be added to support physics like force, mass and velocity.

References

Kleinschmager, S., Hanenberg, S., Robbes, R., Tanter, E. & Stefik, A. (2012). *Do static type systems improve the maintainability of software systems? An empirical study.* Program Comprehension (ICPC), 2012 IEEE 20th International Conference on.

Passos, Eric B., Weslley S. Sousa, J., Gonzales Clua, Estaban W. & Montenegro, A. (2009). *Smart Composition of Game Objects Using Dependency Injection.* UFF ACM Computers in Entertainment, 7(4).

Wenderlich, R. 2013. Introduction to Component Based Architecture in Games. *Ray Wenderlich Tutorials for Developers & Gamers* [web log]. Retrieved Dec, 2014, from http://www.raywenderlich.com/24878/introduction-to-component-based-architecture-in-games

West, M. 2007. Evolve Your Hierarchy. *Cowboy Programming* [web log]. Retrieved Dec, 2014, from http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/

Links

Get the Ludum Engine	**404**
Mono Game Framework	http://www.monogame.net

Credits and thanks

The Ludum engine and this documentation was created by:

Viktor Lyresten
Josef Holmér
Anton Berg
Jacob Nilsson

It was created as an educational assignment for the System Architecture program at the University of Borås. It was developed and documented during November and December 2014.

Thanks for everyone who helped develop the engine by asking questions and criticizing the ugly parts. And thanks to those who answered the questions that made it better.