AAGEngine: Action Adventure Game Engine

Alex Caldas Peixoto
Universidade Tuiuti do Paraná
Faculdade de Ciências Exatas
Curitiba, Brasil

Heitor Murilo Gomes
Pontícia Universidade Católica do Paraná
Programa de Pós-Graduação em Informática
Curitiba, Brasil

Abstract

The game industry has grown a lot in the last years, as a consequence there is a need to develop new games more efficiently. To cope with this problem algorithms commonly used in the development of specific game genres were unified, effectively creating the concept of Game Engines. This work presents the development of new multiplatform 2D Game Engine, namely AAGEngine, written in C++ with SFML library. The main goal of AAGEngine is to provide the key features necessary to develop a 2D action adventure game. The set of features available on AAGEngine includes: character representation; user input control; pathfinding; collision detection; maps manipulation; sound and music handling; and many more.

Keywords: Game Engine, 2D Game, Action Adventure Games.

Author's Contact:

alex.caldas@gmail.com
hmgomes@ppgia.pucpr.br

1 Introduction

As the game industry grows, new game development resources are needed. This includes multimedia libraries, engines, frameworks, and others. Each one of those has different objectives, but the main goal is shared among them, which is: facilitate game development. This work presents the AAGEngine[1] (Action Adventure Game Engine), which was developed with the goal to facilitate the development of a 2D action adventure game. To achieve that goal, the AAGEngine provides two main packages. The first package is denominated Core, and it provides a set of common features necessary to develop a 2D game, such as sprite management, pathfinding, collision detection, hardware access and others. The second package, denominated Adventure, is specialized for action adventure games, thus it provides features like map management, item management, level transition management and others.

The AAGEngine was developed using C++ and uses the SFML [Haller et al. 2013] library for hardware access. The SFML library was chosen because it is available for several operational systems and provides many functions, such as: text rendering, audio manipulation, image handling, window management, and others. The Core package provides generic functionalities of a 2D game engine, and to control the access to the hardware. It facilitates the creation of code that uses sprites, pathfind, collision, and others. We designed the Core package so that other packages, which provide specific features, can be build on top of it. The Adventure package provides common elements of a 2D action adventure game. Some of these specific features are: object, item and map management. There is an integration between the Adventure package features, for example, it is possible to have an item on an object, to represent items and objects on the map and other integrations.

Most existing game engines do not include a specialized layer for specific game genres. On one hand, this characteristic of other game engines provides generality for the solution, since they impose little to no bounds to the development team. On the other hand, it can slow down the development of games with specific characteristics that could be otherwise grouped together. Besides providing a general game development layer (Core package), the AAGEngine also includes a specialization for action adventure games (Adveture package).

The rest of this paper is organized as follows. Section 2 contains a brief description of related works. The Section 3 describes the AAGEngine structure and its modules. Finally, Section 4 concludes the paper.

2 Related Works

Currently, there are many engines, frameworks and graphic libraries available for game developers. Each one of them provide access to functions at different abstraction levels. To develop the AAGEngine a research was made on libraries, engines and frameworks. The analysis had the objective to evaluate the existing features as the implementation of such. For example, OpenGL allow primitive graphics drawing, while SFML, which uses OpenGL and other libraries, provides access to higher level functions that allow the developer to handle audio, network, sprite manipulation and others. The Angel2D, which is a game engine, provides some features specific for game development, such as pathfind, actors and physics.

Haaf's Game Engine (HGE). HGE [HGE 2015] was developed using DirectX 8® [Parberry 2001] and is a fairly old engine, but the HGE has an active community and provides several features for game development. Its first advantage is to be optimized for older hardware. The HGE main features include: GUI with menu elements; Particle system; Resource and multimedia file management; and access to input devices.

Angel 2D. The Angel 2D [ANG 2015] Framework is constantly being updated[2] and contain a vast number of features, including: a Lua interpreter; Physic algorithms; Pathfinding; and others.

Simple DirectMedia Layer (SDL). SDL [Pendleton 2003] is a library that provides access to hardware functions. It is widely used to develop 2D games, but not limited to that. Its main features include: Support to OpenGL® shaders; Textures managements; Text rendering; among others.

Simple Fast Multimedia Library (SFML). AAGEngine uses SFML library to access hardware functions, such as input control. It is possible to use only SFML to create an entire game, but many specific features must be implemented as SFML is a general purpose library.

There are many other 2D and 3D game engines, such as Unity [Felicia 2013], Cocos2D [Itterheim and Lw 2011], libGDX [Oehlke 2013], etc. The Adventure package, on the AAGEngine, is a layer that aids on the development of a 2D action adventure game and similar genres, such as Role Playing Games (RPG) and Real Time Strategy (RTS) games. Effectively, the developer does not need to implement elements, such as: item subsystem; pathfinding; controllers for menus; and others subsystems implemented on AAGEngine. Table 1 presents a summarized comparison of the engines/libraries

[1] Available at: https://github.com/AAGEngine/AAGEngine

[2] The last release on the GitHub® is from 2014

analysed. Since providing an specialized engine for 2D action adventure game development is the main contribution of our proposal, the comparison is focused on aspects that are relevant to this genre.

	AAGEngine	HGE	Angel2D	SDL	SFML
GUI	X	X	X	X	X
Sound	X	X	X	X	X
Sprite	X	X	X	X	X
Collision	X	X	X		X
Pathfind	X		X		
Scripting			X		
Map	X				X
Object	X				
Item	X				
Character	X		X		
Physics			X		
Debbuging	X		X		
Particle		X	X		X

Table 1: *Comparative table*

3 Engine Implementation

This section focus on the description of the AAGEngine inner workings and overall organization. AAGEngine was elaborated aiming at easing the development of 2D action adventure games. To achieve this objective, several features that are usually necessary for this type of game were implemented, for example, item management, collision, map management, and many others. AAGEngine was developed using C++ and the SFML library for hardware access. SFML use several other libraries for specific tasks, the more relevant libraries for the project are listed below:

- Freetype[3]: Manages true type fonts (TTF), allowing to render texts on the screen;
- Libjpeg[4]: Allows the SFML to manage JPEG images;
- Stb_image[5]: Allows loading JPEG, PNG, BMP and other image formats as textures;
- OpenAL Soft[6]: Enables sound files loading. Concretely, this allow the developer to position a sound on a 3D environment;
- Libsndfile[7]: Allows loading WAV, OGG and FLAC sound files format;

The AAGEngine was tested on Windows® and Mac OS X®. Although, theoretically, it is possible to compile AAGEngine on any environment given that SFML and a C++11 are supported. The AAGEngine features were organized into modules, which represents one or more classes. The modules are grouped into two packages, Adventure and Core.

The modules of the Core package were developed with the goal of providing generic implementations to develop a 2D action adventure game. The Core package do not depend upon the Adventure package, thus it can be used independently. The Adventure package uses the Core package to create specialized features that are relevant to a 2D action adventure game.

[3]http://Figura.freetype.org/
[4]http://Figura.ijg.org/
[5]nothings.org/stb_image.c
[6]http://kcat.strangesoft.net/openal.html
[7]https://github.com/erikd/libsndfile

3.1 Core Package

The Core package is responsible for abstracting some of the hardware access to the Adventure package, the only features that access the hardware directly from the Adventure package is the Sound Effect and the Background Music modules. This allow to substitute most of hardware calls from AAGEngine by only modifying the Core package. For example, a developer can replace the rendering calls from SFML to OpenGL and it would not be necessary to modify the Adventure package. It is possible to use the package Core as a foundation to develop a new Adventure package for a different genre of a 2D game. For example, it is possible to use the Image module on the Core package to implement the particle effects of a space shooter game and another features. The modules on the Core package are separated as:

Input Control. The module Input Control provides access to events triggered by the keyboard, mouse and joystick.

Game. The Game module was developed to provide a template or "model" from where it is possible to start the development of a game. This module is implemented as an abstract class, which provides methods responsible for processing the inputs and actions (process), image rendering (render) and a main window object. It is possible to completely ignore the game class while using the AAGEngine. The developer has the freedom to use every module of the Core and Adventure package based on a custom implementation of a game class.

Image. The Image module is responsible, through the SFML library, for loading images to memory, and allowing the developer to manipulate it. Manipulations include: rotation, scaling or changing the position of the image on the screen. It is possible to render the entire image or a fraction of it. The latter is used to render Sprites and Tiles. The file formats supported are: PNG, JPG and BMP.

Collision. The Collision module provides two algorithms to check collision, Axis Aligned Bounding Box (AABB) and Oriented Bounding Box (OBB). The collision area is represented by a bounding box structure, which specifies the object position, size and rotation angle. The AABB checks the collision between two collision areas evaluating the position and the size of the two objects. The OBB works on a similar way, but it also evaluates the rotation angle of the two objects.

Text Control. On action adventure games text handling is important, usually on this type of game, there are several conversations. The Text Control module allows to manage texts and load fonts on TTF format, using the SFML. The module allows to render the text anywhere, on the screen, and to separate the text into smaller portions, this way is possible to correctly render the text within the window resolution limitations.

Pathfind. The Pathfind module have the goal to provides a variation of the A* algorithm, known as the Clearance Pathfind[Harabor and Botea 2008]. The module receives, as an input data: an adjacency matrix; the size of the object; the initial; and the final point of the path. The module creates an internal graph and each node of the graph represents an element on the matrix. Also, each node contains the information of how many adjacent nodes are empty (Clearance level), allowing elements of different sizes to travel through the graph. The module output is a list containing each node that needs to be followed.

Log Manager. Provides to the developer a module for log management. This module allows the user to associate a type of message to an output. The developer can use this module, for example, to output error messages to a file and on screen.

Text Table. The Text Table module was developed with the goal to store all the texts of the game. The texts are

stored in tuples (Key/Value). It is possible to load the texts from a file, those need to have the text stored on the format "Key: text", the "EOK" (End of key) defines that the text of that specific key ended. This module do not provides I18N[8] features, but is possible to archive that using a prefix on the key, such as Name_US.

3.2 Adventure Package

The Adventure package is responsible for specific features needed for an action adventure game. Most engines implements only generic algorithms in order to be as general as possible. The AAGEngine is more specific, since it implements features such as Map Management, Items and other features that are expected on an action adventure game.

Level Manager. The Level Manager module is responsible for handling the active level on the game, and the transition between them. On most games, Levels represent scenery with limited size, on memory and on screen. The main goal of the Level Manager is to make the transition between Levels easier and intuitive for the developer. The Level Manager, like the Game module, is an abstract class with two main methods: process and render. To process and render the active Level, the Level Manager uses the process method, which invokes the process method implemented on the Level. This process happens, also, on the render method. The Level Manager contains a Level object, which is the active level. When a Level change is requested to the Level Manager, it halts the execution of the process and render methods of the currently active Level, and removes it from memory. After that the new Level is allocated and the processing and rendering methods are resumed.

Level. The Level module defines the implementation of one type of scenery. The Level is an abstract class which contains two abstract methods, process and render, those are invoked by the Level Manager. One Level is responsible for the management of a scenery that contains similar execution logic. This allows the same Level implementation to be used to load many Levels with similar logic. For example, a Level with a house, without battle system, is able to load several different maps that, as well as the house, do not need the implementation of a battle system. The Level Manager is responsible for instantiating all Levels as a Level object.

Warp. The Warp module defines locations, inside the map, where it is possible to change to another Level. The Warp module contains the size of the Warp, a file that should be loaded when the player collides with the Warp, and the destination coordinate of the player in the new map. The Warp module cannot change the Level by itself, thus the developer needs to detect the collision between the Player and the Warp and use the Level Manager to complete the operation.

Sprite. The Sprite module is a specialization of the Image module. It was developed with the intent of mapping a sprite-sheet to an in-memory matrix. This process facilitate the access of individual sprites in the sprite-sheet. The image mapping to a matrix format is created when the Sprite module is instantiated. The module requires the image path, width and height of each Sprite. Every sprite on an image must have the same size.

Character. The Character module defines the characters that are not controlled by the player, i.e., Non-Player Character (NPC). The NPC can be anything that is “alive” inside the game, like a merchant or a dog. The module contains a collision area, item list, and an attribute list. In order to represent the Character on the screen a Sprite object is used. The character provides two different methods, with different goals, to animate a character. The first is an automatic movement, which will make the character follows a path passed as a parameter. The attributes of the automatic movement method are: number of pixels that the Character needs to move per second (addPixelsPerSecond) and the nodes that represents the path the Character will move. This path can be generated using the Pathfind module. The Character moves through the path at the speed defined by the addPixelsPerSecond parameter until it reaches the end of the path. The second option is invoke the method “call”. On each execution of the method the module increases the counter on the variable callsOnFrame, when it reaches the value defined on changeOnCall the module will change the frame of the Sprite, progressing to the next frame. The second option is useful to control the movement inside a main loop, where the speed is usually 30 or 60 FPS (Frames per second).

Player. The Player module is a specialization of the Character module and represents the main character. As usual, the main character is controlled by the player, thus a method was created to associate keyboard and joystick inputs to the player’s movement. Any other functionality, besides the movement, must be implemented by the developer.

Map Manager. The Map Manager module manages all the components inside the map on the game, including the 2D camera. The module is compound by several subsystems, which are classes responsible for each one of the elements on the map. Concretely, the map subsystems are: Tile Manager; Tile Layer; Collision Map; Object Map; Item Map; Sound Map; and Warp Map. These subsystems must be instantiated by the developer separately and, as a parameter, passed by reference to the Map Manager.

Object. The Object module represents the inanimate elements inside the game that are possible to interact with, for example, doors and chests. An Object is associated to a Sprite and a list of Items. The Object contains a bounding box to detect interactions between the Player and the Objects on the scenery. The module provides an attribute list, a dictionary, that can be used the way the developer wants, allowing actions such as damaging the player, when it opens the chest, associate "life points" to an object, as another behaviours. The usage of those attributes and the behaviour are responsibility of the developer as the correct visual representations for those interactions. It is also possible to associate an Item to an Object, allowing to the Player to acquire an Item when interacting with the Object. The Objects are managed by the Object Manager, allowing to add and remove an Object, manage the properties of an existing Object and get the ID of the Item associated to the Object. The module allow the Object to be loaded from a file.

Item. Action adventure games have their logic intrinsically associated to the Items scattered through the map. Items may allow the Player to advance the game or simply change its attributes. The visual representation of an Item is possible through the Sprite module. A dictionary is used to map attributes of individual items. Items are managed by a class named Item Manager, which allows additions, removals and selective changes to properties of an existing Item. The collision behavior whenever an Item and a Character collides are specified by the developer. Finally, it is possible to load items from a file.

Menu. Games exhibit different menus, but two main types are usually observed, the configuration and the game Menu. The configuration Menu provides a way to change the configuration parameters, e.g., video preferences. The game menu permits the player to access items, skills, spells, etc. To facilitate the creation of menus, the Menu module provides individual controls that can be added to the screen and rendered on the same way as a Sprite. The actions associated with each of the elements respond to events, executing methods specified by the developer through function pointers or lambda functions.

Sound Effect. The Sound Effect module allows to create a

[8] Internationalization

better ambience on the game, accentuating specific elements on the scenery. The Sound Effect utilizes the Sound class of the SFML, allowing to load uncompressed WAV files. The module allows to control the general volume as the volume on each channel of the sound, it is also possible to pause and restore the reproduction of the audio. Also, it is possible to specify the 2D position of the audio. The Sound Effect Manager is responsible for controlling Sound Effect instances loaded on memory. This management includes: loading sound effects files; creating new instances of sound effects; and adding/removing sound effects.

Background Music. The Background Music is responsible for the creation of a generic ambience. It is possible to have, as an example, a dark song when entering a cave or a happy song when entering a store. The main difference between the Background Music and the Sound Effect is that only one Background Music is allowed at a time. This is because the decompression process of an OGG or MP3 file need a more intense usage of the hardware. Also, this modules is responsible for transitioning between background music with transition effects between them, such as Fade-Out and Cross-fade. Other features of this module includes: pausing/playing songs; checking song's position; reporting song's total time; and controlling the volume.

Dialog. The Dialog module manages the text that represents chats between two Characters. The management of the text is done using the Text Control module. Dialogs are stored on a text vector, allowing to linkages between them. The exhibition of texts respects the order in the vector. This module also allows to create Dialogs with options, which can trigger actions that can be executed using a callback method.

4 Conclusion

In this work, we have presented a game engine with a specialized layer for action adventure games. Because of the specialized layer, the AAGEngine helps to develop an 2D action adventure game, abstracting the common features of this genre. The AAGEngine have two packages, Core and Adventure. The Core package is responsible for abstracting the hardware access, to provide generic features for game development and to be the basis of the Adventure package. The Adventure package provides features specific for action adventure games and almost every hardware access of the Adventure package is done using the Core package, only the sound access is done directly. Because of the AAGEngine architecture, related to the hardware access, is more simple to migrate from one technology to another. The reason for that is because most of the hardware calls are located on the Core package. Most of the modules inside the AAGEngine can be used separately. This provides freedom to the developer on the game development process, because it do not dictates where those modules need to be used. The modules on the Engine package can be used to extend the AAGEngine's features, or it can be used as the basis for other genres (e.g. RPG).

As future works we are considering implementing pixel-perfect collision, because the sprite collision is determined by its size even if there are transparent pixels. The pixel-perfect collision solves that problem, analysing if the collided area has pixel that are transparent or not. To be able to change the behaviour of some elements on the game without recompiling it, the engine should be able to interpret commands from a script at runtime. Some elements of a game can be stored inside serialized files, e.g. maps, items, characters, etc. A level editor provides a way to manage those files visually, thus it would make the process of creating levels, positioning items, objects and others easier. Finally, we are considering porting AAGEngine to iOS and Android platforms.

References

2015. Angel 2d. `http://angel2d.com`, July.

Ericson, C. 2005. *Real Time: Collision Detection*, 1 ed. Morgan Kaufmann Publishers.

Felicia, P. 2013. *Getting Started with Unity*. Packt Publishing.

Gregory, J. 2009. *Game Engine Architecture*. Wellesley, Massachusetts: A K PETERS.

Haller, J., Hansson, H. V., and Moreira, A. 2013. *SFML Game Development*. Packt Publishing.

Harabor, D., and Botea, A. 2008. Hierarchical path planning for multi-size agents in heterogeneous environments. *IEEE*.

2015. Haaf's game engine. `http://hge.relishgames.com`, July.

Itterheim, S., and Lw, A. 2011. *Learn Cocos2D Game Development with iOS 5*, 1st ed. Apress, Berkely, CA, USA.

Kelly, C. 2012. *Programming 2D Games*. Wellesley, Massachusetts: A K Peters/CRC Press.

Oehlke, A. 2013. *Learning Libgdx Game Development*. Packt Publishing.

Parberry, I. 2001. *Introduction to Computer Game Programming with Direct X 8.0 with Cdrom*. Wordware Publishing Inc., Plano, TX, USA.

Pendleton, B. 2003. Game programming with the simple directmedia layer. *Linux journal*, 110, 42–45.