GARTS: Technical Design Document

version 0.0.1 (last edit: 05/12/2016)

Samuel cumming, 1101899

Contents

[Preface 3](#_Toc468827973)

[Version Breakdown 3](#_Toc468827974)

[Technology Discussion 3](#_Toc468827975)

[Unity 3](#_Toc468827976)

[Pros 3](#_Toc468827977)

[Cons 3](#_Toc468827978)

[Discussion 4](#_Toc468827979)

[Unreal Engine 4 4](#_Toc468827980)

[Pros 4](#_Toc468827981)

[Cons 4](#_Toc468827982)

[Discussion 4](#_Toc468827983)

[GameMaker: Studio 4](#_Toc468827984)

[Pros 4](#_Toc468827985)

[Cons 5](#_Toc468827986)

[Discussion 5](#_Toc468827987)

[Appollo 5](#_Toc468827988)

[Pros 5](#_Toc468827989)

[Cons 5](#_Toc468827990)

[Discussion 5](#_Toc468827991)

[Technical Goals 6](#_Toc468827992)

[Core 6](#_Toc468827993)

[Additional 6](#_Toc468827994)

[Technical Risks 6](#_Toc468827995)

[Major 6](#_Toc468827996)

[Minor 6](#_Toc468827997)

[Tools 6](#_Toc468827998)

[SDL 2.0 6](#_Toc468827999)

[Microsoft Visual Studio 2015 6](#_Toc468828000)

[Appollo 7](#_Toc468828001)

[Program Flow 7](#_Toc468828002)

[Overall Flow 7](#_Toc468828003)

[Genetic Algorithm Flow 8](#_Toc468828004)

[Graphics 8](#_Toc468828005)

[Coding Style 9](#_Toc468828006)

[Naming Conventions 9](#_Toc468828007)

[Comments 9](#_Toc468828008)

[Source Control 9](#_Toc468828009)

[Artificial Intelligence 9](#_Toc468828010)

[Non-Learning 9](#_Toc468828011)

[Learning 10](#_Toc468828012)

[Chromosome 10](#_Toc468828013)

[Fitness evaluation 10](#_Toc468828014)

[Document Changelog 10](#_Toc468828015)

# Preface

Genetic Algorithm Real-Time Strategy (GARTS) is an application designed to test whether or not emergent, dynamic artificial intelligences can be more engaging than traditional rules-based AIs in real-time strategy games, resulting in enhancing a game’s replayability.

GARTS comes packaged with, and is built with, the lightweight development framework ‘Appollo’, which was designed specifically for this project. The ‘Appollo’ framework is open-source, and may be used by anyone for developing applications. Currently the framework exists only to support the development of GARTS, and uses SDL2.0 as a backend, programmed in C++.

The following document details the underlying technical concepts required in creating GARTS, and will be updated regularly with changes in the project’s design as they are encountered. This document offers a description of implementations, program-flow, technology decisions along with reasoning, and will discuss chosen implementation routes. This document is intended to be kept up to date and to show project progression as problems are encountered and solutions found. Critical changes will be highlighted to aid in comparison with previous versions of this document. The details laid out in this document are all subject to change as the project progresses and is tweaked.

## Version Breakdown

On the front of this document, a version number is presented (format: #a.#b.#c).

* #a: The release version number. Reflects the current public release. Will be incremented on each major public release of the project.
* #b: Major version number. Reflects the current state of the project in terms of major updates. Will be incremented when a major feature has been implemented.
* #c: Minor version number. Reflects current state of the project in terms of minor updates. Will be incremented when tweaks or bug-fixes are committed to the project.

# Technology Discussion

GARTS will use a custom engine (‘Appollo’) during its development. Other engines were considered, each with their own pros and cons, and are discussed below. Choosing the most appropriate technology during development is essential both for a steady project development and deployment when the project is complete, and so was considered carefully.

## Unity

### Pros

* User friendly UI and IDE, ideal for hassle free development.
* Comes pre-packaged with physics, audio, input and graphics systems ready to be used out of the box, no set up required.
* Deploys to a range of platforms (Windows, Mac, Linux, Android Mobile, Apple Mobile Devices…)
* I have a lot of experience using Unity through previous projects, so there won’t be too many hills to climb.
* Uses C#.Net, which I’m really familiar with.
* Well supported, active community. Ideal for discussing problems with.

### Cons

* Closed Source (i.e Black Box), unable to see underlying engine code for modifications if needed.
* Does a lot for you. Graphics, audio, input are all ready to go from project start, so a lot of the ‘meat’ of the engine is abstracted away from the user. While this is good for some projects, a technical project such as GARTS may require engine tweaks to get the most out of the project.
* Has difficulty with Source Control (GIT) due to Unity’s built-in source control solution. Merging can cause errors most of the time.

### Discussion

Unity is a great engine, however it simply does too much for you, which may impede the smooth running of the project’s development if anything engine-side needs to be tweaked. Unity offers both 2D and 3D rendering, though only 2D is required for GARTS. The engine uses a mix of C#/JavaScript and Unity Script, and I would like to keep the main development language to C++ and utility languages (such as XML, JSON) to keep the project from becoming unmanageable.

## Unreal Engine 4

### Pros

* Extremely powerful IDE, coded in C++. Links to Visual Studio.
* Open-Source, able to go into engine code and tweak if needed.
* Offers a powerful 3D renderer and 2D option.
* Multiple beginner projects/examples to aid in development.
* Very active community, supportive.
* Exceptional suite of tools included, including sprite animator, blueprint scripting and audio cueing
* Git compatible.

### Cons

* Very complex. I’ve had issues with Unreal Engine 4 in the past with it being difficult to use.
* Not designed for 2D game development.
* Designed for First Person Shooter games.
* Better suited for First Person Shooter development.
* Community is fragmented across different versions of the software. A lot of obsolete tutorials/code fragments online.

### Discussion

Unreal Engine 4 is another extremely powerful engine, capable of achieving stunning graphics and physics simulations. However, it was not designed for 2D game development, and therefor is unsuitable for the project. Accompanying this, the engine itself is extremely complex (while open-source, allowing for tweaks if necessary), and it can be quite difficult to find out what’s wrong without a lot of experience with the engine. While I do have experience with it, it simply isn’t enough to justify UE4’s potential risks.

## GameMaker: Studio

### Pros

* Designed for 2D game development
* Contains a complete suite of tools focussing on 2D development (Sprite Editor, Animator…)
* Exceptional help and support, and a very active community of both indie and hobby developers.
* I have a lot of experience using GameMaker: Studio and GameMaker Studio 2.
* I have a master license for GameMaker: Studio, allowing me to export the project to multiple platforms easily.
* Drag & Drop, easing code complexity.
* Uses GML/JavaScript.
* Support staff are generally very active and available.
* Git compatible

### Cons

* Closed Source, so no engine tweaks.
* Drag & Drop simplifies the underlying implementation too much, no class structure available for abstracting code into specific classes.
* As of early January 2017, GM:S will become obsolete in favour of the newer GMS 2 (which is not fully released yet and so not a candidate for engine choice).

### Discussion

GameMaker: Studio is often seen as *the* 2D development tool for computer games. I have a lot of experience with the engine and the benefit of having a good relationship with YoYo Games, the engine’s creator. The engine over-simplifies the implementation to the point where it can actually become difficult to abstract core features (such as the AI, Genetic Algorithm, etc) and may harm future development, or development during the project’s lifetime. It also does not support C++, which is a core requirement of the project.

## Appollo

### Pros

* Custom made framework, specifically for GARTS.
* Open Source, easily extendable.
* Built from the ground up by myself, so I’ll know exactly what it can and can’t do.
* Built in and supports C++
* Uses SDL as a backend framework for graphics, audio, input polling and file-loading.
* Git compatible.

### Cons

* Built from the ground up, may take some time to complete the framework.
* May be difficult to export to platforms other than Windows.
* If the framework will not support everything natively. I’ll have to program additional features and support as problems are encountered.
* Could be difficult to set up on computers other than my own, and may be considered non-portable because of this.

### Discussion

‘Appollo’ is a framework that allows rapid development of 2D games. It uses SDL as a backend for graphics, audio, polling and system commands, and is complete programmed in C++. This will allow for easy code abstraction and complete, full control over the project’s implementation. The framework itself will progress alongside GARTS, and features/support will be added as necessary. The framework will be constructed completely from the ground up, allowing me to gain intimate knowledge of what the framework can and cannot and will help to decrease the complexity of development.

# Technical Goals

## Core

* Simple 2D graphics.
* Animating sprites, for user feedback.
* 1 map with resources.
* User input using mouse.
* Simplistic, easy to use UI.
* Rules-based, deterministic AI opponent.
* Emergent AI opponent.
* Successful implementation of genetic algorithm for Emergent AI.
* Basic combat system.
* Basic resource system.

## Additional

* Basic audio implementation.
* Additional maps.
* Additional AI players.

# Technical Risks

## Major

* ‘Appollo’ may be underdeveloped and hinder project development.
* Genetic algorithm may not have enough time to produce a successful AI opponent during gameplay.
* SDL2.0 may not support a core feature somewhere down the line that hasn’t been considered yet.

## Minor

* ‘Appollo’ may not be as efficient as hoped and may result in some game slow-down.

# Tools

## SDL 2.0

“Simple DirectMedia Layer (SDL) is a cross-platform development library designed to provide low level access to audio, keyboard, mouse, joystick and graphics hardware via OpenGL and Direct 3D.” – Quoted from the SDL website ([www.libsdl.org](http://www.libsdl.org)).

SDL will allow the project to access all of the necessary system resources to display graphics, play audio, amongst others, without the need for a lengthy and complicated implementation into ‘Appollo’. SDL will handle back-end requirements, such as file-loading, and the graphics/audio pipeline, leaving ‘Appollo’ to handle game-logic and the main development environment. It is cross-platform and supported by many developers.

## Microsoft Visual Studio 2015

VS 2015 will be the core IDE used during the project lifetime. The IDE allows full control over the project’s implementation and offers useful tools such as an in-depth debugger, profiler and IntelliSense system. VS 2015 will greatly streamline the development process.

# Appollo

‘Appollo’ is a light-weight, open source and extendible game development framework designed exclusively for the GARTS project. The framework will handle all game-logic and will communicate with SDL for file-loading, input polling and graphics rendering. The framework will be created in C++ and will be abstracted away from the main GARTS implementation. It is expected that ‘Appollo’ will help streamline the development process due to its lightweight “only what’s needed” feature set. The ‘Appollo’ framework will be developed alongside GARTS and extended as needed to support GARTS development.

# Program Flow

## Overall Flow

The following diagram illustrates general program flow from application launch to its termination. Critical functions have been abstracted into separate flow-diagrams.

C:\Users\Sam\Downloads\GARTSOverall.png

## Genetic Algorithm Flow



# Graphics

GARTS requires simply graphics to illustrate resources, buildings and units. Units will be represented by 32x32 sprites, resources 16x16 and buildings 64x64. SDL supports both DirectX and OpenGL drawing, however these are not be necessary since we will be drawing only 2D sprites, rather than primitives or 3D geometry. This will help reduce project complexity and keep the focus of development where it matters, on the AI.

# Coding Style

## Naming Conventions

Camel case (camelCase) will be used when declaring variable names, combined with Hungarian notation for easy identification of variable data types.

Functions will use camel case (CamelCase) with the first letter of each word capitalized. This will help ease of recognition between functions and variables.

## Comments

‘TODO’: Inline commentary that will advise on tasks to be completed. Should be kept as close to the relevant areas in code as possible.

‘BUG ####’: Shows that a bug is suspected of originating from this particular area of code.

‘REFACTOR’: Should be used when a particular area of code requires significant changing, either to be more readable or to reverse a hack.

# Source Control

The project will use Git source-control throughout its development. Commits headers must be concise and descriptive. Commit comments may be whatever is necessary to describe the commit in more detail.

The project’s source control must be kept clean of unnecessary files to help prevent source control bloat.

Although this is a one-man project, a consistent effort must be made to maintain a professional source control history so when the project is released it is easily readable by those unfamiliar with the project.

# Artificial Intelligence

## Non-Learning

The non-learning AI will be presented first to players. The AI will have a predictable pattern and operates in a ‘rules-based’ fashion.

The AI will constantly poll for its progress throughout the game, with major milestones triggering a change in AI behaviour. The non-learning AI has several milestones throughout the game:

* Found new resource – Assign farmers to it (If no available farmers, build one and send)
* Encountered enemy unit – Send nearest unit to attack. If possible, send hard-counter unit)
* Encountered enemy structure – Send nearest unit to attack.
* Reached threshold time limit – Send a scout wave of units to determine enemy positions.
* Resources stockpiled – The AI has gathered significant resources to build powerful units. Build units until resources are exhausted.
* Population cap reached – Build houses to expand population cap, prioritise this.

The non-learning AI will be capable of engaging enemy units and being a competent opponent, however should not be overly challenging.

## Learning

The learning AI will use a genetic algorithm to control its behaviour throughout the game.

### Chromosome

Below is a selection of genes (collectively making up an AI’s chromosome). This is a brief selection of genes, however this table aims to give some insight into how the AI will develop and what values are tracked.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Raiding party size | Preferred Knight ratio | Greed | Building density | Workers per resource | Archery priority | Attack frequency |
| 6 | 33% | 45% | 10 | 3 | 45% | 45 |
| Determines how many units will make up a raiding party. | Prefers army consist of 33% knights, 66% other units. | Determines how likely the AI is to directly go after enemy/unclaimed resources. | Determines how close the AI constructs buildings to each other. | Determines how many workers the AI assigns to a particular resource. | Determines how often the AI will build the archery building. | The time in seconds between attacks against enemy bases. |
|  |  |  |  |  |  |  |

### Fitness evaluation

Each AI will have a fitness assigned to it will which will indicate how likely it is to be selected for recombination. The following values will determine fitness. Each value is core to RTS gameplay, so it’s critical that the AI is assessed via these values.

1. Resources gathered: The total number of resources gathered. An AI that is underperforming in resources may stagnate at the late-game stage and falter due to a lack of resources. By monitoring this value, we can ensure that selected AIs are definitely resource self-sufficient and expanding competitively.
2. Units constructed: An army is crucial to victory in GARTS, so the AI will have its unit construction value monitored. Higher values will indicate that the AI is performing well in military preparation and using collected resources adequately.
3. Damage dealt: An army is one thing, but proper management and unit composition is also crucial. AIs that perform well by having a range of units (or a mass of one particular unit that counters enemy units successfully) will be considered fitter and is likely to be more successful.

The values will be compared against other AI values via a ‘Roulette Wheel’, with fitter candidates taking up more space on the roulette wheel, being more likely to be selected for recombination.

# Document Changelog

(05/12/2016): Initial document draft completed.