Masters in Informatics Engineering
Thesis/Internship
Final Report

A Language for Game Design and Coreography

Nuno Barreto
nbarreto@student.dei.uc.pt

Professor Adviser:
Licínio Roque
Date: July 3rd of 2013

FCTUC DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Abstract

This final report details a proposal approach to model concurrent actors' behaviors and choreographies in the context of video games and simulations, using Petri Nets as a base for its modeling language. Petri Nets have already been used to model some video-game aspects such as game design because they provide a simple and easy way to learn to model interactive systems as well as a means of model validation, through simulation.

This proposal is complemented by an architectural conceptualization that allows the run-time editing of said language, alongside its execution on a pre-existing game engine. As such, it involves a visual editor application, a Petri Net simulation/execution mechanism and an interoperable communication structure that permits linking the editor, and execution module, with an arbitrary game, or simulation, engine.

To validate this proposal, a proof of concept game prototype and usability tests were conducted issuing positive results.

Keywords

"Complex Systems" "Behavior Modeling" "Game Design" "Simulation" "Software Architecture" "Petri Nets" "Visual Language"

Table of Tables

Table of Figures

Table of Terms

Term	Definition
AAA Games	Video-games developed by a large studio (with a team bigger than 100 people), usually funded by a millionaire budget.
Actor	Game object capable of expressing behaviors (see Behavior).
Actor Archetype	Class of actors that have a well-defined collection of behaviors.
Actuator	Type of motor used to move or control a system.
Agile Development	Set of software development methodologies built upon iterative and incremental processes.
API	Application Programming Interface. Source code based library which includes functions and data structures.
Atomic action	One or more actions that must be completed all at once, otherwise they are cancelled. Therefore, on a macro-level they count as one.
Behavior	Set of actions made by an entity.
Behavior Model	Data structure detailing how a behavior functions.
Bipartite graph	Graph whose vertices can be divided into two independent disjoint sets in which edges link a vertex from one set to another in the opposite set.
Cache	An in-memory representation of a hardware cache where items are stored for later access.
Choreography	Visual and audio manifestation of behaviors (see Behavior).
Code Polish	The act of algorithm optimization and software debugging (see Software Debugging).
Complex Systems	A science field that studies how systems function as a whole given their parts.
Crowd Behavior	Group of actors behaving as if they were one (see Behavior).
Design Space	A collection of all possible design choices for a given element.
Development Toolkit	A group of assets and scripts used to make games of a certain genre (see Script).
Directed graph	A graph whose edges have an associated direction.
Framework	A platform that can be used to develop applications. It may contain APIs (see API).
Game Engine	A system that abstracts machine specific tasks in order to simplify the creation of a game.
Game Mechanics	One or more rules that define how gameplay, of a particular game, works (see Gameplay).
Game Middleware	Software that provides a set of functions, above the OS layer, to simplify the act of building game engines.
Gameplay	The way a player interacts with a game.
Genetic Programming	Family of algorithms inspired in the biological phenomenon known as evolution.
GUI	Graphical User Interface. An interface which contains visual elements. (see Interface).
Interface	A software bridge between the user and the application.
Lag	Slow response of a computer application caused by a faulty internet connection or lack of computational power.
Language Semantic	The meaning, in terms of computation, of a language construct.
Language Syntax	The collection of rules that define how the language's symbols are displayed.
MMO	Massive Multiplayer Online. Online games/simulations containing a large number of simultaneous players.
Module	A part of an application with well-defined responsibilities and services.
Path Finding	Family of algorithms that aim to find the shortest path between two points.

Program	Set of steps that transforms given inputs into outputs.
Script	A fraction of software code with a set of commands or configurations.
Sensor	A mechanism that measures a given aspect.
SOAP	Simple Object Access Protocol. A network protocol for interoperable information exchange.
Software Architecture	The high-level specification of how the software, as a system, works: how its modules communicate with each other and how the data flows. (see Module).
Software Artifact	Software, or a software part, resulting from the development process.
Software Debugging	The act of finding and reducing software defects. (see Software Defect).
Software Defect	Part of an application that does not function as intended.
Stage	The game world in which actors can interact and navigate (see Actor).
Visual Language	Programming language that relies on manipulation of graphical elements instead of writing textual commands.

Index

1. Introduction

Video-game development is a multidisciplinary area which can encompass diverse skill set such as game design, programming, sound engineering, 3D modeling or even 2D illustration. From the aforementioned disciplines, programming has an important role in the development as it translates the game's concept, as envisioned by the game designer and interpreted by the artists, into an interactive multimedia artifact known as a video-game.

Since game programming is what contributes to the creation of aspects including how game objects - referred from now on as actors - are modeled in an environment and, likewise, how they interact with each other and with their environment, it becomes necessary to translate every possible course of action into the actor's code which can be an error-prone task.

With the rise in project complexity in AAA games and with the increased accessibility of tools for small independent teams to create games, simplifying the creation of game code provides an added value for developers in terms of code maintenance and polishing, giving them more time to perfect the game rather than trying to fix its code.

This document proposes a solution to simplify the process of game programming which involves the creation of a visual language based on Petri Nets. More specifically, this language is intended to model concurrent actors' behaviors and choreographies among the game world, referred to as stage. Therefore, the project underlying this document is composed of the language's specification, a visual editor where petri nets can be created and edited and an interpretation engine that translates modeled behaviors into in-game actions on an arbitrary game, or simulation engine through a communication system.

This solution is not only interesting for video-games but also simulations, more specifically, that of complex systems. The proposed tool offers a new approach to model actor behaviors in both video-games and simulations.

Petri Nets have been proven accessible and easy to learn by non-programmers and offer an economical way of specifying behaviors in complex systems. As a design tool, they give the advantage of a simple visual language that promotes agile modeling and testing of complex interactive systems such as video-games.

The remainder of this report is structured as follows: Section 2 will depict the state of the art regarding the usage o Petri Nets in modeling video-game related specifications, Petri Net editors, approaches to develop video games, ways to define actor behaviors and choreography, and distributed architectures that support Massive Multiplayer Online (referred to as MMO) games. Section 3 will state the document's overall research methodology and work plan. Section 4 will display, and explain, the proposed architecture for the project's systems. Section 5, will describe the application's implementation and its consequent revision of the architectural proposal. Section 6, will explain the evaluation used and its consequent analysis. Section 7 will detail the future work. Finally, Section 8 will conclude the dissertation and reveal its lessons learned.

As appendixes, this document also includes an example of a work backlog and defect list (A), the design doc and task-list used for the usability tests (B and C respectively), the tests' recorded issue compilation (D), the raw data gathered from the usability tests (E), User-by-Problem (F) and Task-by-Problem (G) matrices, also collected from the tests, and a table detailing color scaled issues according to their priority levels (H).

2. State of the Art

2.1 Video-game Software Development Approaches

2.1.1 Game Engine Definition

According to Ward (2008), the definition of a game engine is a system whose purpose is to abstract common (and sometimes platform dependent) computational game related tasks that include rendering, physics and input from the game's specific logic. Engines exist to allow developers to focus on their game rather than recreating said tasks.

Ward (2008) also states the existence of three types of game engines: the low level engines (composed of libraries and middleware), mid-level engines and high level engines - He differentiates the two latter by the amount of code required in order to create a game, being none in the case of high level engines.

In this report, I will refer to low level engines as libraries and game middleware and mid-level and high-level engines as game engines.

2.1.2 Approaches to Game Development

During the course of years, the way video-game software is created has evolved. No longer are programmers required to build the game's software components from scratch as they were until the late 80s - as explained in Ward (2008).

When developing video-game software, developers have mainly two options: they either build their own game engine or they use a preexisting engine as the bottom layer for their game logic code. Both ways usually require that developers have some programming knowledge or at least a programmer on their team.

2.1.2.1 Coding From "Scratch"

Although not literally from "scratch", this approach requires developers to build their own engine which increases the effort in making a video-game. In it, developers can opt to create their engine by building its own components such as physics, AI, rendering and input management or integrate their code with existing libraries, often called "Game Middleware", which contain common routines for said components. This usually gives more implementation freedom than using a pre-made engine, as the developer can code exactly what he wants, but it may require some additional knowledge (algebra, physics, etc...) depending on how low level the implementation is going to be. Another disadvantage is "integration hell". This is caused when "Game Middleware" has faulty documentation, is poorly coded, or it is simply not suitable to be integrated with other components or middleware.

2.1.2.2 Using a Preexisting Game Engine

Most of the time, due to time constraints or lack of skillsets, building an engine from the ground up is not a viable option, so using a preexisting one provides a better approach on such cases. Nevertheless, this method of building games has two main disadvantages: some engines are too expensive for independent developers and others are well suited for particular types of games, requiring an additional effort to make them support other genres.

Most engines still require some programming knowledge (as will be described in subsection 2.2) in order to be used.

In this category of game development approaches, lies modding tools. Modding tools are applications that allow modifying several aspects of an existing game (hence the name "modding"). These tools are usually packed with games and can be manipulated in a similar manner as game engines are. Therefore, the usage of modding tools can be considered as using a specialized engine.

2.2 Development Tools

Widely used on Linux ported and academic games, SDL (Simple DirectMedia Layer) is a library created in C that provides a cross platform low level access to computer components such as audio, keyboard, mouse, network and 3D hardware through Open GL (a cross platform library also written in C but, with bindings for other languages, that gives access to the graphics card in order to render 2D and 3D images). This library, much like Open GL, supports bindings for a wide variety of languages such as Java, ADA or Lisp.

Table 1 SDL Specs

Characteristics	Explanation
Purpose	Multimedia C/C++ low-level access library.
Key Concepts	Functional programming and low level resource management (memory, etc...).
Platform	Linux, Windows, BeOS, Mac OS X, FreeBSD, NetBSD, OpenBSD, BSD/OS, Solaris, IRIX, and QNX
System Requirements	Any system, as long as its graphics card supports OpenGL.
Languages	C, C++, Ada, C#, D, Eiffel, Erlang, Euphoria, Go, Guile, Haskell, Java, Lisp, Lua, ML, Objective C, Pascal, Perl, PHP, Pike, Pliant, Python, Ruby, Smalltalk and Tcl.
Visual Editor Characteristics	It does not contain a visual editor.
Game Genres	Any game genre.
Licensing Model	The library can be used in commercial applications as long as it is access through dynamic link.
Behavior Modeling	Behaviors can be freely coded in any way as the library does not support an API for behavior modeling.
Multiplayer	Provides an external library called SDL_net to build for networking, but it can be integrated with any other library.

Microsoft's XNA Game Studio is a framework used to create video-games for Microsoft's platforms which encompass Xbox, Windows and Windows Phone. It is, however, mainly employed to create Xbox games. XNA was written on top of Direct X (Microsoft's collection of APIs for multimedia development), and uses C# .NET as its native language. Similar to SDL, it contains access to network, input, audio and image rendering.

Table 2 XNA Specs

Characteristics	Explanation
Purpose	Game Engine.
Key Concepts	Object Oriented Programming.
Platform	Windows, Xbox and Windows Phone.

System Requirements	Windows XP or later; Graphics card that supports Shader Model 1.1 and DirectX 9.0c.
Languages	C#.
Visual Editor Characteristics	It does not contain a visual editor.
Game Genres	Any 2D and 3D game genre.
Licensing Model	Free for commercial and non-commercial PC games as long as they do not use GFWL API. Using GFWL API for commercial games requires a special agreement with Microsoft. Xbox commercial games requires that developers join the App Hub and pay an annual fee of 99$ as well as a 30% of revenue for Microsoft.
Behavior Modeling	Behavior can be coded.
Multiplayer	XNA contains GFWL network API that can be used. Other libraries such as .Net framework network API can also be used.

Regarding browser based games, the Flash technology is still a popular choice. This technology allows to easily create vector images, import images and videos and create animations using key-frames, however to create more complex systems such as video-games, one must still program using Flash's integrated language, ActionScript, which contains functionalities similar to those found in SDL and XNA.

Table 3 Flash Specs

Characteristics	Explanation
Purpose	Multimedia authoring tool.
Key Concepts	The screen in flash is called a stage and the entities are seen as actors that interact on said stage. Developers must be familiar with grouping objects through parenting. Developers must be familiar with animations created with the use of key frames.
Platform	PC, Mac, Android and some Tablets.
System Requirements	Intel Pentium 4; Windows® XP with SP3 or later or Mac OS X v10.6.8; 2GB of RAM; 3.5/4(for Mac) GB HDD space; A Monitor that supports 1024x768 screen resolution; JRE 1.6; QuickTime 7.6.6; Graphics card with at least 64MB of VRAM compatible with DirectX 9.
Languages	ActionScript.
Visual Editor Characteristics	The editor contains a timeline where developers can create animations or add events. Developers can drag and drop objects onto the stage. Editing some of the actors' parameters (mostly visual parameters such as size, etc...) can be done through textboxes and combo boxes.
Game Genres	Any 2D or 3D game genre.
Licensing Model	The software costs 699$ and can be used to create commercial applications.
Behavior Modeling	Behavior can be coded.
Multiplayer	ActionScript provides a network socket-based API and additionally Adobe Flex can be acquired as it also provides a network API.

A rising substitute to the aforementioned technology is HTML5, an updated version of HTML, supporting, for instance, playing videos and rendering vector graphics. Akin to Flash, building video-games using HTML5 relies on external languages such as Javascript.

Table 4 HTML5 Specs

Characteristics	Explanation
Purpose	Language for presenting structured WWW content.
Key Concepts	Developers must be familiar with tag based languages.
Platform	Any browser.
System Requirements	Any computer that supports a browser.
Languages	Javascript and PHP.
Visual Editor Characteristics	It does not contain a visual editor.
Game Genres	Any 2D and 3D game genre.
Licensing Model	Free to use for commercial and non-commercial applications.
Behavior Modeling	Behaviors can be coded.
Multiplayer	Multiplayer is achieved through APIs such as Websocket, ExpressJS, NodeJS, etc...

Another Multimedia tool that is widely used for 2D games is Scratch. But unlike Flash, Scratch contains a visual language (Resnick, et al., 2009). This language works by stacking blocks of instructions to create a program sequence. Instruction blocks contain written commands, similar to that of textual programing languages, comprehending, for example, "if" and "for". Each block has arguments that can be filled in the form of combo or text boxes. Not all blocks stack together; instead they work in a manner analogous to puzzle pieces. Scratch, unlike most tools presented throughout this document, is oriented towards teaching computation and mathematical concepts as well as acting as a collaborative platform for developers to create projects cooperatively (or "remixing" projects as defined by Scratch authors).

Table 5 Scratch Specs

Characteristics	Explanation
Purpose	Multimedia and programming learning tool.
Key Concepts	Functional programming based on command block stacking.
	The screen in scratch is called a stage and the entities are seen as actors that interact on that stage.
Platform	Windows, Mac and Linux.
System Requirements	Display supporting 16 bit color; Windows 2000 or later, Mac OSX 10.4 or later, Ubuntu 9.04 or later; 120MB of HDD space; speakers and microphone.
Languages	Scratch.
Visual Editor Characteristics	The editor contains a stage where developers can drag and drop "actors". By clicking on the actors or the stage itself, developers can drag-and-drop predefined command blocks and stack them to form program sequences that model stage and actor behaviors.
Game Genres	Any 2D genre.
Licensing Model	The software is free to use.
Behavior Modeling	Behaviors can be coded. It contains predefined sensors, making it easy to develop reactive agents.
Multiplayer	Contains an API for network called Scratch connections.

There are many tools that can be used in order to avoid coding every game engine component. One of the most popular is the collection of tools provided in Havok Behavior.

These tools add computations for physics, AI and animations. They can be used as APIs but they also allow some visual state-machine editing.

Table 6 Havok Behavior Specs

Characteristics	Explanation
Purpose	Dynamic character behavior development tool.
Key Concepts	Developers must be familiar with state machines.
Platform	PC and Consoles.
System Requirements	Not found.
Languages	C++, Lua and Havok Script.
Visual Editor Characteristics	The visual editor contains a viewport where developers can observe the animation results. There is also a window where developers can create their state machines by linking squares (states) through arcs. Editing parameters is done through textboxes and checkboxes.
Game Genres	Any 3D Game genre.
Licensing Model	Havok must be contacted prior to acquiring licensing as well as the software.
Behavior Modeling	Developers can organize entities' animations in states (each state containing an animation) using its visual editor. By adding events and linking states through arcs, developers can create state machines that model how the entity animates given a particular situation.
Multiplayer	Not applicable.

Box2D is a 2D physics API for Javascript, with bindings for other languages. It provides simulation for constrained rigid bodies by acting forces upon them. Compound bodies can be built by linking other, simpler, bodies. This framework also has the possibility to simulate gravity, friction, restitution and collision detection.

Table 7 Box2D Specs

Characteristics	Explanation
Purpose	2D Physics Engine.
Key Concepts	Object Oriented Programming.
	Basic knowledge in rigid bodies, torque, impulse and forces.
Platform	Windows, Mac, Flash and Linux.
System Requirements	A system running the supported platforms.
Languages	C++, Flash, Java, C# and Python.
Visual Editor Characteristics	It does not contain a visual editor.
Game Genres	Not applicable.
Licensing Model	The software is free to use for non-commercial and commercial applications.
Behavior Modeling	Physics behavior can be coded using the provided API.
Multiplayer	Not applicable.

AgentSheets is a visual design environment developed for the purpose of programming dynamic agents in simulations and other applications (Repenning & Citrin, 1993). The way this environment works is based on spreadsheet applications, where cells can contain formulas that affect neighboring cells. It contains a grid called worksheet where agents can be displayed among its cells (Rausch, 1998). Here, agents can affect, or be affected by, neighbor agents. Behaviors are modeled through rule based systems using the provided language, AgentTalk. AgentTalk is similar to Scratch in the way that it is composed of instruction blocks that can be dragged and dropped. Parameters can be altered through combo boxes and other GUI components.

Table 8 AgentSheets Specs

Characteristics	Explanation
Purpose	Tool for the creation of agent-based games and simulations as well as a computer science learning tool.
Key Concepts	Developers must be familiar with formula creation in a spreadsheet.
	Developers must be familiar with rule based programming.
	Content addition through brush-painting.
Platform	Windows and Mac
System Requirements	A system running the supported platforms.
Languages	AgentTalk.
Visual Editor Characteristics	The editor contains a grid where developers can drag and drop agents and other objects. There is also a window containing a tile-set of said agents or objects.
Game Genres	Any 2D genre.
Licensing Model	The software is available for commercial use for 97€. There is also a free 3-trial available upon user registration.
Behavior Modeling	Behaviors can be coded.
Multiplayer	Not applicable.

Finally, the last tool presented in this subsection is ToonTalk (Kahn, 1995, 1996a, 1996b, 2000, 2006). ToonTalk offers a different way to model agent behaviors, and overall system programming, than any of the tools mentioned in this subsection. It is an application aimed for younger children to learn computational concepts through the use of metaphors and examples. ToonTalk works like a sandbox video-game where the player's avatar lives in a city in which he is prompted to solve particular problems through the construction of programs. As was mentioned earlier, ToonTalk relies on metaphors to illustrate computer science concepts: the city, the avatar lives in, represents a computation; houses represent agents; birds represent network systems. Players must manipulate these metaphorical objects in order to build programs inductively, or by "programming by example". This means that players start by writing program sequences to solve particular problems and then try to generalize those sequences. Program flow can be observed through the animation of the metaphorical objects.

Table 9 ToonTalk Specs

Characteristics	Explanation
Purpose	Programming learning tool.
Key Concepts	Programming by example.
	Programming through animation.
Platform	Windows.

System Requirements	A system running the supported platforms.
Languages	ToonTalk.
Visual Editor Characteristics	The editor is made in the form of a city where each city component such as houses, etc… represent a computational abstraction.
Game Genres	Mostly puzzle games.
Licensing Model	Free for commercial use since version 3.0.
Behavior Modeling	Behaviors can be coded.
Multiplayer	Not applicable.

2.3 Engines

UDK (the "free" version of Unreal Engine 3) is an engine primarily made for first and third person shooters. In order to develop games using UDK, developers can script game objects' behaviors using its property language called Unreal Script, or use the Kismet tool (a visual scripting tool, that was made, essentially, for level design). Besides providing an easy simple way to develop scripting events, Kismet allows for visual debugging, so developers can observe the script's flow. The main disadvantage of using Kismet is that its coded behavior is only valid on the level it was created, meaning that games built with this tool having more than one level, may require most of the visual code repeated throughout its levels.

Table 10 UDK Specs

Characteristics	Explanation
Purpose	Game Engine.
Key Concepts	Optimized levels can be created directly on the editor using the supported BSP geometry.
	Building games in UDK works by modding the built-in game.
Platform	It builds games for PC, Mac, IOS, Flash, Android and Consoles.
System Requirements	Windows XP SP2 or Windows Vista ; 2.0+ GHz processor; 2 GB system RAM; SM3-compatible video card; 3 GB free hard drive space.
Languages	UnrealScript for scripting behaviors and C++ for editing the engine's source code.
Visual Editor Characteristics	Contains a level editor that allows modeling simple geometry which serves as a skeleton for the level's architecture. It also allows to drag and drop entities onto the level.
	Entities' properties and parameters can be edited through textboxes and checkboxes.
	Level scripts can be created using a visual language, Kismet, that works by linking constants (circles) and functions (square) together through arcs.
	UDK also contains a material editor that works much like Kismet, by linking algebraic operations and constants together through arcs in order to build shaders.
	It also contains a cut scene editor that has a timeline where developers can add/remove key frames and events and preview their cut scenes.
Game Genres	The engine is best optimized for level-based shooters, although with some effort other genres can be built with it.
Licensing Model	UDK is free for educational and non-commercial use.
	To use the application within a business, an annual 2500$ fee is required per developer seat.
	For commercial applications, a 99$ fee is required upfront and an additional 25% of revenues after the application reaches 50000$ in revenues.

Behavior Modeling	UDK contains a class called "AIController" that has methods for decision making and basic character functions where behaviors can be coded using UnrealScript. UDK also supports Nav Mesh and Node Waypoints placed on the level editor for path finding.
Multiplayer	Provides an API to build multiplayer games with a server based architecture using RPC and state synchronization.

Ogre is a cross-platform graphics engine that uses C++ as its coding language. Unlike other engines mentioned in this subsection, this one has no visual tools and, therefore, is only manipulated through code.

Table 11 Ogre Specs

Characteristics	Explanation
Purpose	Graphics Engine
Key Concepts	Object Oriented Programming.
Platform	PC, Mac and Linux.
System Requirements	A system that supports Open GL or Direct X
Languages	C++
Visual Editor Characteristics	It does not contain a visual editor.
Game Genres	Any 2D and 3D game genre.
Licensing Model	The software is free to use for non-commercial and commercial applications.
Behavior Modeling	Behavior can be coded.
Multiplayer	Multiplayer can be achieved by using an external library such as Raknet.

Much like Ogre, Flixel is an engine that contains no visual tools and must be manipulated through code. It was built with ActionScript and uses that language as its native. Both Flixel and Ogre are general purpose engines, meaning that they are prepared for the development of any type of game.

Table 12 Flixel Specs

Characteristics	Explanation
Purpose	Game Engine.
Key Concepts	Object Oriented Programming.
Platform	Flash.
System Requirements	Flash system requirements.
Languages	ActionScript.
Visual Editor Characteristics	It does not contain a visual editor.
Game Genres	Any 2D game genre.
Licensing Model	The software is free to use for non-commercial and commercial applications.
Behavior Modeling	Behavior can be coded.
Multiplayer	Multiplayer can be created using external libraries and platforms such as Smart Fox Server.

Unity is a general purpose engine that, like UDK, is comprised of several visual tools. It also contains a propriety online store called asset store, where developers can download/buy various third party and official plugins including the "platformer kit", a toolkit made to build generic platform-based games, which makes coding not required for simpler games. Nevertheless, to develop more complex video-games, programming is required and Unity supports 3 different scripting languages: Boo (based on Python), UnityScript (based on Javascript) and C# .Net.

Table 13 Unity Specs

Characteristics	Explanation
Purpose	Game Engine.
Key Concepts	Entities are seen as game objects that can be extended by adding one or more components (scripts) in order to give them behaviors (such as physics, animations, AI, etc...). Since it is a multigenre game engine additional developer tools can be made or acquired from the asset store. It provides an API for developers to create their own visual editors or extend the ones that come with Unity. Collision layers for physic optimization.
Platform	It builds games for PC, Mac, IOS, Flash, Android and Consoles.
System Requirements	Windows XP SP2 or later; Mac OS X: Intel CPU & "Snow Leopard" 10.5 or later; Graphics cards compatible with Direct X9 and with Occlusion Query support for PC and Mac games. Android OS 2.0 or later; Device powered by an ARMv7 (cortex family) CPU; GPU support for OpenGL ES 2.0 for Android Games.
Languages	Boo, C# and UnityScript for scripting components and C++ for editing the engine's source code.
Visual Editor Characteristics	Unity comes with simple 3D level editor where developers can drag and drop objects onto it. It also comes with visual editors to modify game object components for developers to edit parameters through text boxes and check boxes and an animation editor with a time-line, where developers can add/remove key-frames for several game objects' parameters and preview their animations.
Game Genres	Any game genre but it is more suitable for 3D games.
Licensing Model	A free version of the game engine is available with restrict functionality that can only make PC/Mac games. To access the cut functionalities, Pro version must be bought for 1140€. To develop for Flash, Android and IOS, an additional license must be acquired for 305€ and 1140€, the normal and pro version respectively for each of the aforementioned platforms. To access the source code or console developing licenses, unity technologies must be contacted .
Behavior Modeling	Unity contains a class called "MonoBehaviour" that has methods called at certain frames (each frame, each physics frame, etc...) where behaviors can be coded using one of the mentioned scripting languages as well as a component called "CharacterController" for basic character functions such as moving. The Pro version also supports Nav Meshes placed on the level editor for path finding.
Multiplayer	Provides an API to build multiplayer games with a server based architecture (Raknet) using RPC and state synchronization. Can use external plug-ins for multiplayer.

Although not as popular for independent games as UDK, CryEngine is also an engine oriented towards the shooting genre. It contains some visual tools, resembling UDK and Unity. More specifically it provides an event scripting tool similar to Kismet. Another interesting tool present in this engine is a behavior tree editor, where developers can create behavior trees easily. This engine also offers a SDK written in C++ that allows developers to extend the engine's capabilities, allowing the creation of other game genres, for example.

Table 14 CryEngine Specs

Characteristics	Explanation
Purpose	Game Engine.
Key Concepts	Developers must be familiar with level editing based on brush painting. Developers must be familiar with flow charts and decision trees.
Platform	PC and consoles.
System Requirements	Windows XP or later; 64-bit CPU; 2GB RAM; Graphics card that supports Shader Model 3.0 or better
Languages	C++ and a visual language similar to C++.
Visual Editor Characteristics	The editor provides "brush paint" terrain editing. Entities' parameters edit is done through textboxes and checkboxes. The engine also provides a flow chart like visual editing to create events and character scripts. It also provides a visual decision tree editor for AI.
Game Genres	The engine is best optimized for shooters, although with some effort other genres can be built with it.
Licensing Model	CryEngine is free for educational and non-commercial use. For independent game developers who want to create commercial games, a licensing model is provided that consists in 20% royalties of the game's revenue.
Behavior Modeling	Behavior can be created using a flow chart that links, through arcs, variables and methods or by creating a visual decision tree.
Multiplayer	Provides an API for client-server multiplayer architecture.

RPG Maker XP, as the name implies, is an engine built for the purpose of creating RPG games. It is more oriented towards classic turn-based isometric 2D RPGs and contains visual editing tools including an event editor and a tile map editor to aid the creation of such games. Building games with core mechanics akin to Final Fantasy is made easy using the provided tools and require no coding effort. Nevertheless, RPG Maker XP supports Ruby as a scripting language and scripts can be used to extend the engine's functionality.

Table 15 RPG Maker XP Specs

Characteristics	Explanation
Purpose	Game Engine.
Key Concepts	Developers must be familiar with building tile-based maps.
Platform	PC.
System Requirements	Microsoft XP or later; Intel Pentium III 1GHz or equivalent; 256MB Ram; A Monitor that supports 1024x768 screen resolution; DirectX compatible hardware; 100MB HDD free space.
Languages	Ruby.
Visual Editor Characteristics	Contains a level editor that allows building multi-layer tile based maps. Developers can "paint" those maps using tiles on a tile set. They can add terrain characteristics such as non-walkable terrain, etc … in a similar painting manner using meta tiles. For events and character/monster edit, etc... RPG Maker provides a parameter editor based on text boxes and checkboxes.
Game Genres	Best optimized for 2D turn based isometric tile-based RPG.
Licensing Model	There are 3 versions of RPG Maker: VX4, VX and XP (with VX4 having the most features and XP having the least). Each version comes with a free trial for non-commercial use. For commercial applications, RPG Maker must be bought. The prices are: 89.99$ for VX4, 59.99$ for VX and 29.99$ for XP.
Behavior Modeling	Behaviors can be coded using the language provided and built-in game classes.

Multiplayer	The application comes with the Windows API and therefore multiplayer can be built using Windows Socket.

Kodu is a 3D multigenre game engine aimed specially for children. It comprises brush painting tools for terrain generation and a parallel rule based system used to create game agent behaviors.

Table 16 Kodu Specs

Characteristics	Explanation
Purpose	Game Engine.
Key Concepts	Developers must be familiar with rule based programming.
	The concept of brush painting for terrain generation.
Platform	PC and Xbox.
System Requirements	Windows XP or later; Graphics card that supports Shader Model 2.0 or later and DirectX 9.0c; .Net Framework 3.5 or higher; XNA framework 3.1 or higher.
Languages	Kodu.
Visual Editor Characteristics	The level editor uses brushes to help developers "paint" terrain.
	Entities can be drag and dropped onto the level.
	Editing the entities' behavior, works by creating a stack of rules and each of them is edited by adding/removing context dependent components. Components are added in a pure visual way by pressing a plus icon which then opens up a wheel containing the available components.
Game Genres	Any 3D game genre.
Licensing Model	Free to build non-commercial games.
Behavior Modeling	Behaviors can be modeled by creating rule-based agents. Each rule uses an <condition><action> syntax. As was mentioned in "visual editor characteristics", these rules can be created by adding components to each member of the syntax. Since the components are context dependent, it is impossible to create invalid rules. In addition, Kodu provides several different sensors that can be used in the rule creation process.
Multiplayer	Only offline multiplayer is supported.

Stencyl is a general purpose 2D engine made to develop IOS and Flash games, incorporating a tile map editor and an event editor, much like RPG Maker XP. In order to create events and game behaviors, developers are required to use Stencyl's visual language, which is Scratch. But, unlike Scratch, the application, Stencyl's goal is only to allow developers with no programming knowledge to create Flash games easily. Stencyl also contains an "asset store" analogous to Unity, where developers can download toolkits to create games.

Table 17 Stencyl Specs

Characteristics	Explanation
Purpose	Game Engine.
Key Concepts	The screen in Stencyl is called a stage and the entities are seen as actors that interact on said stage.
	Functional programming based on command block stacking.
	Searching for available game toolkits as to not "reinvent the wheel"
	Developers must be familiar with building tile-based maps.
Platform	IOS and systems that support Flash.
System Requirements	No system requirements were found.

Languages	Scratch.
Visual Editor Characteristics	Contains a level editor that allows building multi-layer tile based maps onto the stage. Developers can "paint" those maps using tiles on a tile set. They can add terrain characteristics such as non-walkable terrain, etc … in a similar painting manner using meta tiles. Developers can also drag and drop "actors" on to the stage. Developers can also drag and drop behavior "blocks" onto the actors or build these blocks by stack command blocks in order to create a program sequence.
Game Genres	Any 2D game genre.
Licensing Model	A free version of the software that can be used for commercial Flash games exists with limited functionality. The extended version costs 79$ per year and to develop commercial games for IOS it is necessary to buy an IOS subscription for 149 per year. Additionally, it is necessary to have an Apple Developer account to test games on IOS devices.
Behavior Modeling	There are available behavior kits that can be used by drag and dropping them onto entities. Additional behavior can be coded.
Multiplayer	Stencyl requires a socket server for multiplayer games and an additional plugin to build the client side code.

The last tool presented in this document is Little Big Planet 2's level editor. This editor can be seen as a game engine as it allows developing a variety of 2D and 3D games. It provides developers building "pieces" such as sensors, reactors, etc..., and logic operators. This way, developers can construct mechanisms by using the editor's primitives or build more complex creations by linking "pieces" together with operators.

Table 18 Little Big Planet level editor Specs

Characteristics	Explanation
Purpose	Modding tool for Little Big Planet.
Key Concepts	Developers should be familiar with simple Boolean logic. Developers should be familiar with mesh creation using "brush" painting. Developers should be familiar with Little Big Planet mechanics such as checkpoints, etc... Developers should know how to use a PS3 controller.
Platform	PS3.
System Requirements	PS3.
Languages	Not applicable.
Visual Editor Characteristics	The editor comes in form of a blank level where developers can open a popup menu to drag and drop objects onto it. The editor also comes with mesh "brushes" to create forms and bolt objects to link forms together. Parameter edit comes in form of a slider or check boxes.
Game Genres	Almost any 3D and "2D" sidescroller game genres.
Licensing Model	To use this tool, a copy of Little Big Planet 2 is required. All levels created, by third party users, using this tool should be non-commercial.
Behavior Modeling	Little Big Planet provides a variety of sensors such as timers, proximity sensors, etc... , logic operations such as AND, OR and NOT and reactors such as movers, shooters, etc... Reactive and rule based agents can be easily modeled using these tools.
Multiplayer	Levels created in this tool are multiplayer levels.

2.4 Approaches to Define Computational Game Behavior

The definition of computational game behavior follows mainly two types of approaches: It is either manually defined by the designer or it is procedurally generated in run-time. Nevertheless, the latter method also involves the creation of a set of rules, or design space definition, that generators must respect in order to generate content. These definitions can be either textual or visual (through the use of diagrams).

2.4.1 Manual Definition of Game Behavior Rules

As was stated earlier, in this set of approaches, behaviors are well-defined by the designer. They can either be an exhaustive set of rules, where every action/reaction is accounted for (which happens for most commercialized games) or one that allows the emergence of behavior as in complex systems.

The most classical approach to define behavior is to do it as a code script. This, or a visual variant of textual coding, is the default manner used in most of the tools presented in 2.2. It consists in defining behaviors using a programming language (the examples shown use either functional or object oriented languages). The main disadvantage of this approach is that the designer must be familiar with computation concepts as well as with the programming language. Nevertheless, this way provides a flexible means to create behaviors and serves as the building blocks to the other mentioned approaches.

Another way is the creation of finite-state machines (Milligton & Funge, 2009). Finite-state machines are a visual approach used to model simple actor behaviors, animation transitions or, generally, game mechanics. It is composed of states and arcs linking them together. Each state comprises a set of actions and each arc a set of conditions. Nevertheless, this approach has scalability issues in which more complex state machines become unreadable and difficult to maintain.

Behavior trees (Milligton & Funge, 2009) are another visual tool used to define behaviors. Behaviors can be modeled by creating a tree where each leaf node represents an action, each non-leaf a condition to be tested and each branch a different outcome for a given condition test. Much like finite-state machines, behavior trees can bloat, thus becoming unreadable and difficult to maintain.

Black box diagrams, present in UDK and CryEngine, are another graph-based tool. In both these engines, this tool works the same way: there are a number of premade function nodes, or "black boxes" (visually represented as squares with several inputs and outputs) and data nodes (represented as circles). A designer can link these nodes in order to model behaviors so that information can flow from outputs, and data nodes, into inputs. Whenever information arrives at a function node's input, a scripted event takes place. The presentation in this tool is analogous to digital circuits diagrams. In addition, much like the other visual tools, black box diagrams have a problem of becoming unreadable as the they grow.

Rule-based systems are a model in which the designer defines a set of condition-action pairs as to design simple behaviors. This allows for emergent behaviors as shown in Baptista and Ernesto (2008). A variation of this model is achieved by using fuzzy logic (Milligton & Funge, 2009) in the condition part of the pair instead. Moreover, this variation is popular in commercial video game systems.

The BDI architecture, or Belief-Desire-Intension, is another approach in modeling actor behaviors. It was created to be akin to how humans make decisions (Woolridge, 2011). In this architecture, an actor has a belief, desire and intention set. Beliefs represent the actor's perception of its inner and environment states, while desires are the actor's preferable end-states and intentions are the actions it chooses. This architecture has additional functions: the belief update function that allows beliefs to be refreshed according to new perceptions, the options function that uses actor's beliefs and intensions in order to produce desires and the filter function in which beliefs, desires and intensions are used to generate intentions. These mechanisms underlying the BDI architecture result in a plan, or a set of actions, for an actor to follow. Designers can model behaviors in this architecture by initializing the actor with different beliefs, desires and intentions. This architecture, in conjunction with PSI psychological theory for the display of emotions, is used in Lim, Dias, Aylett, and Paiva (2009), with the purpose of creating autonomous NPCs in an educational RPG.

Cellular automata (Milligton & Funge, 2009) are another manner used to model behaviors. Cellular automata are grid structures representing world space that are iteratively updated. Each cell's content is renewed according to a function that takes into account the content inside a given cell's neighbors. This approach is well suited to model crowd behaviors as demonstrated in Bandini, Manzoni and Vizzari (2004), and for the design of emergent behaviors as the iterative nature of cellular automata cannot be easily predicted.

The approach used in Little Big Planet 2's level editor also provides a different way to model behaviors. The editor supplies the designer with different types of sensors, actuators and logic functions. By combining sensors and actuators, through logic functions, designers can create different behaviors.

Dormans (2009) created a framework, called Machinations, which supports the definition of game play mechanics. This framework accentuates the view that games are rule-based dynamic systems. It provides a diagram tool so that designers can elaborate their game mechanics (which include behaviors). More specifically, these diagrams represent the "flows and foreground feedback structures that might exist within a game". As such, this tool provides a means to illustrate game mechanics that are otherwise difficult to enunciate as they are hidden in the game's system. The author argues that this framework, unlike other tools, for instance, finite state machines, is accessible for designers and well-suited to represent "games at a sufficient level of abstraction". The main disadvantage of this framework is that it is not appropriate to design games that rely on level design because it does not take into account all elements of game design.

2.4.2 Procedural Content Generation

This approach offers a different perspective on the responsibility of computers in playing interactive media. Here computers also act as dynamic content providers instead of being the traditional content players. The dynamic nature of this approach allows for customized experiences where each play-through is different.

One example of such approach is the interactive drama Façade (Mateas & Stern, 2005a , 2005b). This video-game tells the story of the rupture of a marriage in a first-person perspective. Unlike commercial games where the narrative is pre-made, in Façade it is emergent. The system reacts to the player's input, whether through actions or speech, to develop plot points at run-time by mixing sequences of narrative.

Genetic Programming is also a method used in procedural content generation. Genetic Programming is a family of algorithms that are inspired in the biological phenomenon known as evolution. Their main objective is to generate programs in the form of a computational tree or graph, that can solve a given problem. Their modus operandi is as follows: a population of candidate programs are iteratively crossed together (known as reproduction) in order to produce offspring. These offspring can also be mutated, thus introducing diversity in the population. The original population and the offspring are filtered using selection mechanisms in which the program candidates are selected according to their fitness (a rank given to the performance of the program). This family of algorithms can be used to generate behavior trees or finite state machines to model actor behaviors as was stated in Kadlec (2008).

2.5 Usage of Petri Nets in Game Specifications

Although public research regarding the usage of Petri Nets in modeling video-game aspects is scarce, there have been some scientific papers describing how Petri Nets can model certain aspects of video-games with success.

Araújo and Roque (2009) showed that Petri Nets could be used to model a game. In their case study Sagres, the game's system and flow was modeled into Petri Nets organized in hierarchies. It was concluded that Petri Nets, as a graphical tool, were more expressive and easier to understand than other tools such as those provided in the UML. The simulation capability of Petri Nets was also viewed as an advantage over other design tools as it brought designers a means to evaluate their artifact.

Natkin, Vega and Grünvogal (2004) introduced a new methodology for temporal representation of levels in video-games. It consisted in using Petri Nets to describe temporal relationships in scripted events, for example, how were the actions a character must perform to achieve certain goals related in terms of their partial order.

Brom and Abonyi (2006), developed a system, exploiting Petri Nets, for the authoring of nonlinear plots and story management in a large interactive virtual reality world populated by a significant amount of virtual actors. This system used Petri Nets to specify a high-level plot detailing all of the possible events and actions that could happen which would alter the flow of the story. This model ran on a simulator, called story manager, that kept track of the plot's course while checking for events to unfold and, consequently, altering the simulation world according to where were tokens. It was concluded that the system functioned as intended (the story manager altered the course of the simulation according to its plot whilst giving the actors independence in their individual behaviors) and provided a means for prototyping and validating plots.

In sum, Petri Nets have been used, by other researchers, to model some video-game aspects such as plot and game design. It was mostly concluded that this tool was easy to learn and understand and provided a means for validation, facilitating prototyping. Additionally, they were also perceived as extensible and, consequently, have a wide variety of augmentations that add functionalities which, in turn, reduces potential model complexity. Because of their apparent smooth learning curve, validation functionality, extension capability and the demonstrated ability to design several aspects of a video-game, this tool was considered as a candidate for the base of the language described in this report.

2.5.1 Petri Net Definition

According to Petri and Reisig (2008) and Wang (2007), Petri Nets are a graphical tool invented, in August 1939, by Carl Adam Petri to describe chemical processes. Generalizing, this tool can be used to describe and analyze concurrent distributed processes (i.e. an interactive system such as a video-game) and their flow (through simulation). Figure 1 illustrates a simple Petri Net.

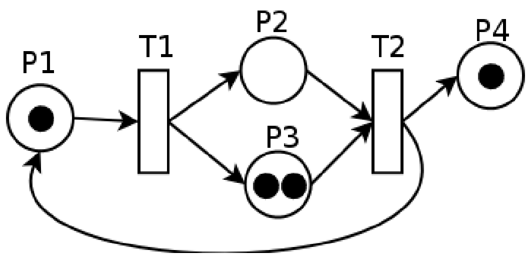


Figure 1 A simple Petri Net

2.5.1.1 Graphical Definition

Graphically, Petri Nets are a subset of bipartite directed graphs composed of four primitives: places, transitions, directed arcs and tokens. Places are circular shaped nodes that abstractly represent conditions. Places can be filled with one or more tokens, a black and small circular object. A token inside a place means that the place's inherent condition was met. Directed arcs are used to link a place to a transition, a black shaped node, or vice-versa. The place at the head of an arc is called an input place and, likewise, a place at an arc's tail is called an output place. Arcs can be weighted, meaning that a certain number of tokens are either required for a transition to fire or that they "travel" through the arc. Non-weighted arcs can be seen as weighted arcs with the weight value of 1.

An important concept associated with Petri Nets is the concept of firing transitions. A transition is said to fire when there are enough tokens at its input places. This leads to the consumption of tokens from input places and the production of tokens at the output places, which can abstractly mean that an event occurred. It is worth noting that firing is atomic, meaning that the consumption and production of tokens regarding a given fired transition is considered one step.

Another important concept is marking. Marking is the name given to the net's configuration represented by the distribution of tokens across places.

2.5.1.2 Mathematical Definition

A more formal definition of Petri Nets is that they can be represented by a 4-tuple $N := (P, T, F, M_0)$, where P and T are sets of places and transitions respectively; $F : \{P \times T\} \cup \{T \times P\}$ represents the flow, i.e. the arcs connecting the net's places and transitions; M_0 is the initial marking.

2.5.2 Petri Net Models and Editors

2.5.2.1 Data Structures

In this context, data structures are computer representations of Petri Nets. This includes in-memory representations or even description languages. Data structures may be part of APIs that provide functions for net manipulation.

There are several Petri Net APIs for a wide variety of languages. They provide similar methods to manipulate Petri Net objects and their main difference seems to be which extensions they support and whether or not they include a simulator. Examples of APIs include JFern, the Petri Net API (Lohmann, Mennicke, & Sura, 2010) and C++ Petri Net Framework (Nadle, 2003).

As an attempt to make Petri Net models interoperable and standard, a description language, called Petri Net Markup Language (or PNML) (Billington, et al., 2003), was created. This language is built upon XML and, besides describing how Petri Nets are structured, it allows to add graphical attributes, which provides visual editors the means to render the models, as well as to append tool-specific attributes that can only be parsed by designated editors.

From PNML was derived EPNML (Werf & Post, 2004), a XML format that is used in Yasper. This format uses elements from PNML and, although its specification is available, it is argued by the author that the syntax definition is incomplete.

2.5.2.2 Editors

There are several Petri Net editors available that can be used to model interactive systems. Most of these tools are aimed to illustrate a wide variety of systems, as their main application is to develop concept diagrams that demonstrate how systems, and workflows, function. Therefore, they are deprived of semantics, which makes their simulation functionalities, if present, mainly for visual debugging purposes. However, there are some editors that allow users to add semantics to their Petri Net models. The following table offers a comparison between Petri Net editing tools.

Table 19 Petri Net Editor Comparison

	Editors										
Characteristics	PNEditor	PIPE	Petri Net Editor	Tina	Yasper	WoPeD	JPetriNet	TAPAAL	PetriNet Kernel	JFern Editor	Snoopy
Petri Net Extensions Supported	Hierarchical .	Colored, Timed and Stochastic with inhibitor arcs.	Stochastic.	Timed with inhibitor arcs.	Hierarchical, Timed with inhibitor arcs and XOR transitions.	Hierarchical with AND and XOR transitions.	Timed.	Timed-Arc with inhibitor arcs.	Hierarchical .	Hierarchical and Colored.	Stochastic, Timed, Colored, Extended, Hierarchical, Music and Hybrid.
Simulation	Manual step based.	Manual or Automatic step based.	Manual step based.	Automatic step based with time ruler.	Manual or Automatic step based.	Manual or Automatic step based.	Automatic step based.	Manual step based.	Manual step based.	Manual or Automatic step based.	Automatic step based.
Semantics	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	Tokens can be associated to java classes.	Transitions can contain Java scripts and tokens can be associated to java classes.	Music Petri Nets' transitions can play sounds on firing.
Export Format	PNML, EPS and PNG.	PNG, PS and eDSPN.	PNG and XML.	PNML, TNP and PNT.	VDX and several Image formats.	PNML and several Image formats.	N/A	PNG, PS and TikZ.	PNML and XML.	XML, ser and jnf.	EPS, Latex, MIF, Xfig, ANDL, APNN and several proprietary formats.
Written Language	Java	Java	Java	TCL/C++	.Net	Java	Java	Java	Java	Java	C++
Additional Functionalities	Allows to add roles to transitions.	Features several analysis modules.	Features several analysis modules, a Markov chain generator and the capability to associate probabilities to transitions.	Features several analysis modules.	Allows to add roles to transitions.	Features several analysis modules and a graphical layout optimizer.	Can build a matrix/tree representing the net.	Features several analysis modules and a query mechanism.	N/A	Can generate Java source code.	N/A

As can be observed, only JFern Editor and, to some extent, Petri Net Kernel and Snoopy allow to add user-made semantics to the net. However, JFern Editor requires the user to introduce code manually. It is worth noting that JFern Editor is an editor using the JFern Petri Net data structure described in the earlier sub-chapter.

2.6 MMO Architectures and Network Middleware

With the increase in popularity of developing massive online multiplayer games, it became logical to extend the initial proposed solution so that it could work with this genre, as it requires for the application to have a special architecture in which distributed computing, and network programming, is needed to provide the means to play the game with various simultaneous online players.

2.6.1 MMO Architectures

There are mainly three types of distributed architectures used in MMO games: Peer-to-Peer, Client/Server and Hybrid.

2.6.1.1 Peer-to-Peer

This type of architecture relies on using clients (or peers) as a means to gather more computational resources such as processing power and memory. Therefore, application data is completely distributed among peers. In fact, in this architectural approach, there is no centralized server.

Hampel, Bopp and Hinn (2006) introduced an architectural proposal using DHT (Distributed Hash Tables), Pastry and Scribe. In this architecture, data is replicated and distributed across peers and indexed on a DHT for lookup. When peers need to access data, they look it up using Patsy and after they retrieve the requested data, it is then synchronized across the network using Scribe. In order to achieve this, peers are given fat clients containing every component necessary to run the game as well as additional components for security and resource management. Anti-Cheating mechanisms are available as the architecture elects peers, as controllers, to monitor manipulated data for inconsistent game states. It is argued that this architecture provides attributes needed to run MMO games such as scalability, load-balancing and traffic optimization. Nevertheless, it is not assured that this architecture provides reduced latency for time critical applications, such as action games.

2.6.1.2 Client/Server

Unlike the previous architecture, the Client/Server relies on a centralized server to process client requests and to act as a message router to relay information among clients.

To increase scalability in a Client/Server model, Müller and Gorlatch (2004) devised an architecture that consisted in the addition of proxy servers to account for load balancing. When a client, connected to a proxy, sends a game state update, it is then processed and multicast across all proxy servers to keep them synchronized. By analytical comparison, the authors concluded that this proposal provided more scalability than regular Client/Server or Peer-to-Peer architectures.

2.6.1.3 Hybrid

This approach delegates some heavy computational responsibility to clients and, at the same time, maintains a main server to secure consistency. It combines the resource availability of a Peer-to-Peer architecture with the security of Client/Server architecture.

Axelrod and Amir (2012) developed a hybrid high-response, low bandwidth architecture for the development of MMORTS (a genre that requires real time responses and management of a significant amount of in-game actors). In it, they explain that some assumptions can be made which optimizes resource consumption: groups of close agents can be processed as a single agent since, statistically, they tend to receive similar commands (an example is walking to position x); messages sent to the server can be compressed; the stage can be fragmented into regions to reduce updates, as players only visualize stage portions at a given time which, in turn, allows to cluster players into said regions; actions of simple agents can be predicted as they are statistically similar over time; AI intensive algorithms can be distributed to the clients to take advantage of the clients' computer resources.

Douceur, Lorch, Uyeda and Wood (2007) also suggest distributing AI algorithms to clients as a means of load balancing. In this architecture AI is split into server-side and client-side components. Server-side AI computations are simple, lag intolerant and do not require client-side computations. Client-side computations, on the other hand, are slow, stateless and complex and may be distributed to several clients as a way to minimize failures. Server-side AI uses client-side AI to improve its effectiveness. The authors determined, through a prototype, that this architecture improved AI abilities in an action game with latencies up to one second.

2.6.1.4 Comparison

In sum, each architecture provides a different approach to accommodate the existence of applications that work across a network. From the research made, it was concluded that each architecture has its strengths and weaknesses. Table 20 shows a comparison made regarding the architectural concepts presented in the former subsections.

Table 20 MMO Architecture comparison

	Characteristics							
Architecture	Implementation Complexity	Client Size	Scalability	Anti-Cheating Mechanisms	Data Consistency	State Consistency	Latency	Bandwidth
Client/Server	Simple, only requires developing a server that receives and sends processed game logic.	Thin	Scalability is proportional to the number of servers.	Servers can check for inconsistent game states sent by clients.	Relevant data is kept on a server-side database and as such is consistent.	Servers must assure that all clients receive the same state updates in the same order.	Latency may be increased as requests/respon ses must go through the server.	Servers require large bandwidths to accommodate requests/respon ses from clients
Peer-to-Peer	Complex, requires to develop clients that sends and receives processed game logic and a means to keep data consistent among clients.	Fat	Scalability is proportional to the number of connected users.	Mechanisms that involve selected users to act as consistency checkers for game state.	Data is spread across users. Therefore, it is required data replication and consistency checks to assure that data is not corrupt	Users must check each other's states for consistency.	Peers can communicate directly with each other, decreasing latency.	Bandwidth is optimized because clients communicate directly with each other.
Hybrid	Medium to Complex, requires developing a server, and clients that can communicate with themselves.	Thin or Fat	Scalability is proportional to the number of connected users.	Servers can check for inconsistent game states.	Data can be spread across users as replicas of the servers' data. As such it is required that servers check for users' data for consistency.	Servers must assure that all clients receive the same state updates in the same order.	May have some latency as some requests must go through the server.	Requires bandwidth to accommodate client/server communication.

2.6.2 Network Middleware

Because network programming is difficult to learn and master, there are several network middleware tools that provided the necessary abstractions so that developers do not need to interact with network programming and, in some cases, develop a distributed architecture in which their game can run. As such, games relying on data exchange across a network, as MMO games do, benefit from such middleware.

Table 21 depicts the comparison among popular network middleware regarding licensing price, supported concurrent users, architecture and other important notes.

Table 21 Network Middleware comparison

	Characteristics			
Middleware	Price ($)	Users supported	Important Notes	Architecture
Smartfox Server	• Free • 500 • 1000 • 2000	• 100 • 100 • 500 • Unlimited	• Unity API • Cloud support • Supports zone fragmentation	Client/Server
Raknet	• Free if revenue is under 100k for version 4 • Pay – per – Application	• Unlimited • Unlimited	• C++ API • Unity comes with version 3 • "Unity's" C# can be used to communicate with Raknet code	Client/Server
Photon Server	• Free • 1250 • 2250 • 3500	• 100 • 500 • 1000 • Unlimited	• Contains a cloud • Business logic can be written in C# • Contains logic for MMORPGs and FPS. • Provides a free Unity asset for networking using the photon cloud	Hybrid
ElectroServer 5	• Free • 999 • 4990	• 50 • 1000 • Unlimited	• Unity API	Client/Server
Badumna	• Pay-As-You-Grow	• Unlimited	• Supports Unity	Client/Server

3. Methodology

This project followed a Design Science Research approach and its produced prototype was developed according to an agile development process.

3.1 Objectives

As written, the main purpose behind this solution proposal was to develop an easy to use modeling tool, based on Petri Nets, that could be used by non-programmers, to define actor behaviors and choreographies in a game/simulation environment, whether it is running on one machine or on a distributed system. Because of the initially stated broad definition of actor (i.e. any game object), this authoring tool could be used not only to define entities' computations but also general mechanics that govern the game/simulation.

Given the solution's objectives, the proposal was devised to contain the visual language's specification (syntax and semantics) as well as a system comprised of three modules: a visual editor, an execution module and a communication module.

This research aims mainly to answer two questions: whether it is possible to build a system in which Petri Nets model actor's behaviors and choreographies and, being possible, if it is easy to use and understand by the intended audience.

3.2 Design Science Research

Design Science Research (Hevner & Chatterjee, 2010) is a methodology that aims to produce a statement of learning as a consequence of research made through design or, in other words, this methodology's objective is to solve problems with the purpose of producing a statement of learning.

The Design Science Research, as illustrated in Figure 2, encompasses 5 steps: Awareness of Problem, Solution Proposal, Prototyping, Evaluation and Statement of Learning, each producing its own artifacts. Since this methodology can be used in any area where design is possible, the following clarification of the methodology's steps states example artifacts best suited in this project's context.

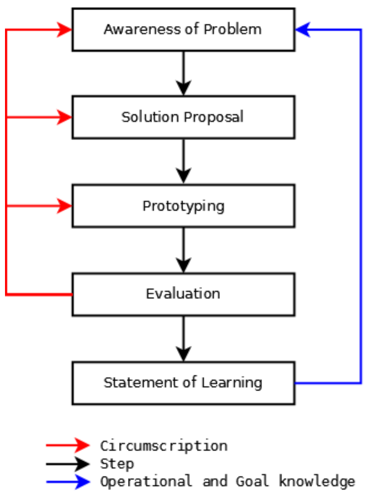


Figure 2 Design Science Research steps

The first step, or Awareness of Problem, comprises the definition and identification of a problem. In order to help clarify this definition and identification, State of the Art research is made.

In the following step, Solution Proposal, a suggestion for the problem's solution is created through abduction drawn from the state of the art research made previously. This step outputs interaction models and software architecture.

Prototyping comprehends the actual development. In this stage, software is produced, using the conceptual models created in the previously. The artifact produced acts as a proof of concept, proving that the proposed solution is possible.

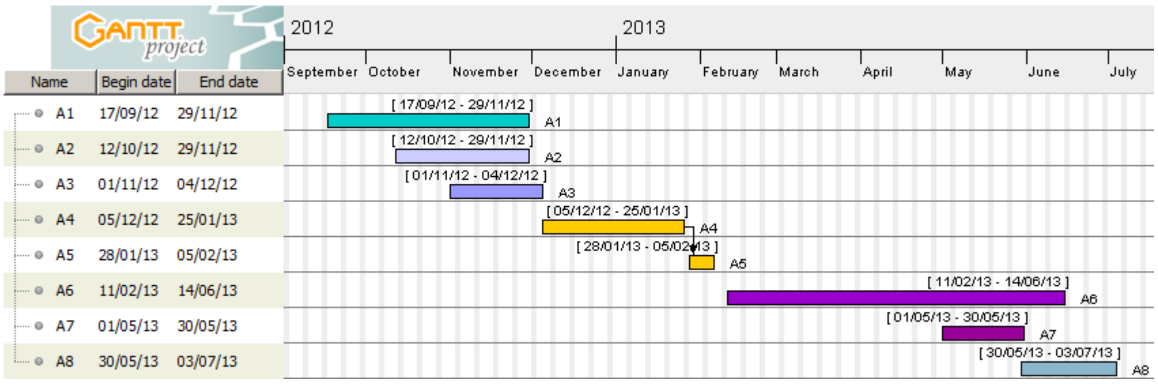
The next step, or Evaluation, attempts to validate the prototype created in the previous stage using an evaluation model. When such model is nonexistent, one is devised alongside other Solution Proposal's artifacts. This step also provides feedback, or circumscription, to the other previous steps which permits an iterative and incremental development (the agile development process is used during the iterations Prototype-Evaluation).

Finally, in the Statement of Learning step, the project is concluded so that knowledge can be produced. Artifacts produced in this stage include concepts, models, methods and prototypes.

This research method is adequate for the project in question because it promotes agile development which ensures refinement of the application to meet the project goals and by following its stages, it is possible to answer both research questions detailed earlier.

3.3 Work Plan

The work plan devised for this project divided in 8 steps, or activities, spread across both semesters. Figure 3 presents the project's expected timeline.



Name	Begin date	End date
A1	17/09/12	29/11/12
A2	12/10/12	29/11/12
A3	01/11/12	04/12/12
A4	05/12/12	25/01/13
A5	28/01/13	05/02/13
A6	11/02/13	14/06/13
A7	01/05/13	30/05/13
A8	30/05/13	03/07/13

Figure 3 Project Timeline

3.3.1 Activities

As was mentioned earlier, the project will be developed with 8 activities in mind:

A1 State of the Art Research. This activity comprises research of relevant SoA regarding software, similar approaches and related work in order to study the problem this thesis intends to solve.

A2 Detailed Proposal. This activity comprises the definition of the problem's scope and respective solution through the elaboration of project goals, requirements and assumptions and the creation of milestones for progression tracking.

A3 Initial Design. This activity comprises the initial draft for the editor's graphical interface as well as the applications' (editor and petri net running engine) proposed architecture and their respective quality attributes.

A4 Mid-Term Progress Report. This activity involves the writing of this progress report.

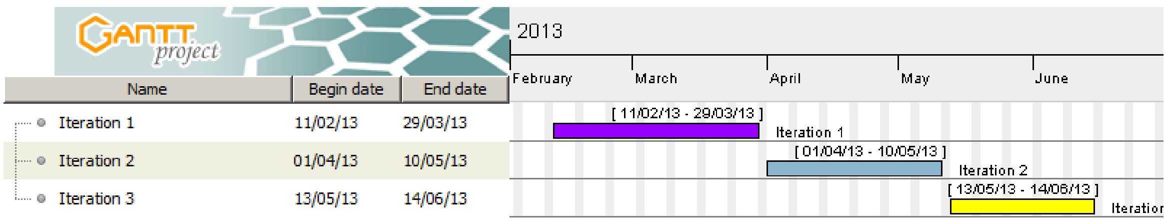
A5 Mid-Term Presentation. This activity, dependent on A4, includes the creation of the project's mid-term presentation.

A6 Prototyping. This activity comprises the implementation and testing of a working prototype of the applications and subsequent fine-tuning based on evaluation results.

A7 Evaluation. This activity comprises the definition of the evaluation methodology and criteria and consequent result analysis.

A8 Statement of Learning. This activity encompasses the production of this thesis' final report and reflection on knowledge production.

To elaborate further, the prototype activity (**A6**) is divided into three monthly iterations, each involving several activities, which aim to produce this document's project prototype. The following figure depicts these activities in a timeline:



Name	Begin date	End date
Iteration 1	11/02/13	29/03/13
Iteration 2	01/04/13	10/05/13
Iteration 3	13/05/13	14/06/13

Figure 4 Prototype Iterations

Iteration 1:

- **Engine choice and GUI Window Implementation.** Activity where the appropriate game-engine and middleware is chosen. It also comprises the creation of the editor's GUI window.
- **Petri Net Model and Persistent Storage Mechanism.** Creation of the Petri Net model and XML file reader and writer so that it can be used to store and load said models. Additionally, the button listeners to import/export Petri Nets to disk are also created in this activity.
- **Petri-Net manipulation Button implementation.** The implementation of the buttons and actions used to add/remove places, tokens and transitions as well as links. This activity also comprises the implementation of a naming mechanism for the Petri Net components.
- **Creation of the Simulation Engine.** Creation of the Petri Net's engine component responsible to simulate transitions.
- **Integration of the simulation engine in the editor.** Integration of the simulation engine in the editor so that it allows to simulate Petri Nets' executions.
- **Validation mechanism implementation.** The implementation of the Petri Net's validation mechanism and button action.

Iteration 2:

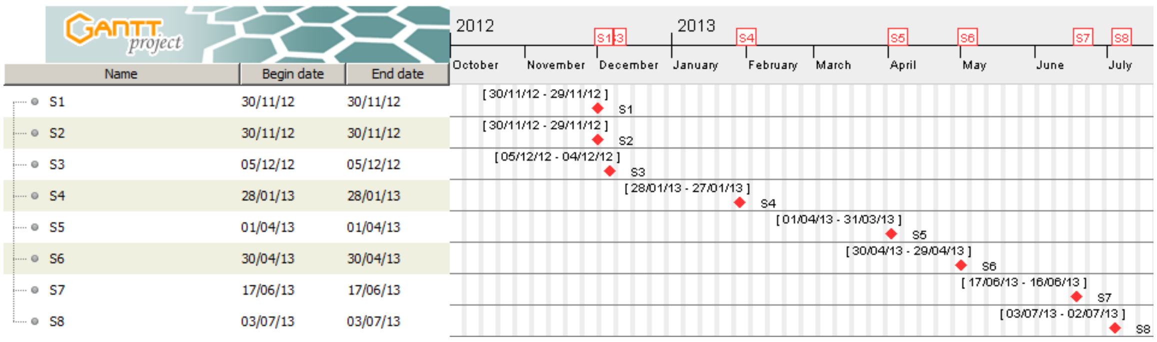
- **Redo/Undo and Petri-Net Grouping Implementation.** Implementation of redo/undo buttons and actions as well as Petri Net grouping and naming.

- **Script integration in the petri-net model.** Integration of script incorporation in places and transitions.
- **Script integration in the simulator.** Implementation of executing mechanisms that allow for scripts to run.
- **Message Queue Implementation.** Implementation of the message queue used to communicate with engine components.
- **Communication with the game engine components.** Implementation of the adapters for the game engine components and consequent API calls.

Iteration 3:

- **Game prototype design.** The design of the prototype that demonstrates this project's abilities.
- **Game prototype implementation.** The implementation of the showcase prototype.

3.4 Milestones



Name	Begin date	End date
S1	30/11/12	30/11/12
S2	30/11/12	30/11/12
S3	05/12/12	05/12/12
S4	28/01/13	28/01/13
S5	01/04/13	01/04/13
S6	30/04/13	30/04/13
S7	17/06/13	17/06/13
S8	03/07/13	03/07/13

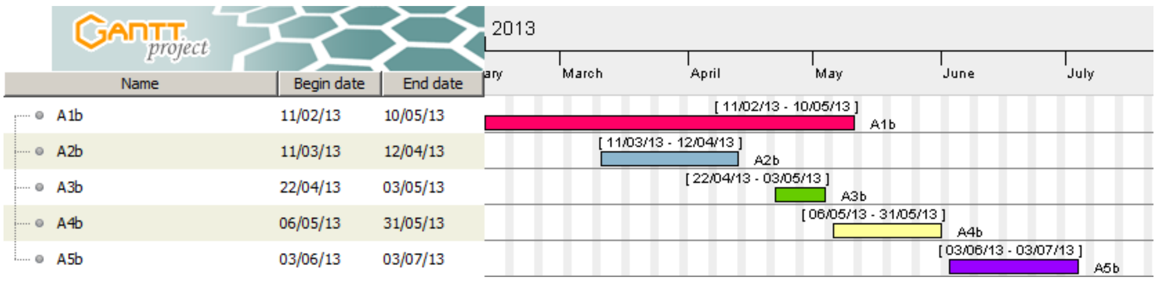
Figure 5 Project's Milestones

This project milestones, as illustrated in Figure 5, are as follows:

S1 Document stating SoA relevant to this project. Due on ***30/11/2012***.
S2 Prototype of the editor's GUI. Due on ***30/11/2012***.
S3 Applications' architectural diagrams (boxed and class diagrams). Due on ***5/12/2012***.
S4 Mid-term progress report. Due on ***28/1/2013***.
S5 Prototype v0.1 comprehending the editor's GUI (with all of the **MUST** requirements implemented), a means to persistently store Petri Nets and the simulation component of the Petri Net's interpretation engine. Due on ***31/3/2013***.
S6 Prototype v0.2 containing the implementation of the editor's **SHOULD** requirements as well as integration of the interpretation engine with the selected game engine and petri-net to game engine code mapping. Due on ***30/4/2013***.
S7 Prototype v1.0 that comprises fine tuning of the aforementioned components and a working prototype that allows proof of concept of the system. Due on ***16/6/2013***.
S8 Final progress report. Due on ***3/7/2013***.

3.5 Effective Work Schedule

During the second semester, the initial planning had to be revised since some activities had discrepancies between their expected time for completion and the actual time they took. Moreover, taking these new times into account, there was enough room for further activities. The following figure details the effective work schedule:



Name	Begin date	End date
A1b	11/02/13	10/05/13
A2b	11/03/13	12/04/13
A3b	22/04/13	03/05/13
A4b	06/05/13	31/05/13
A5b	03/06/13	03/07/13

Figure 6 Effective Work Schedule

A1b. **Prototyping**. This activity comprises the implementation and testing of a working prototype of the applications (further details are explained in section 5).

A2b. **Proof of Concept Development.** This activity details the development of the game prototype that served as the application's proof of concept.

A3b. **Writing of Scientific Paper.** During this activity, an initial draft of a paper explaining the application and its architecture proposal was written.

A4b. **Evaluation**. This activity covers the definition of the evaluation methodology and criteria, consequent result analysis and application's refinement according to the results.

A5b. **Statement of Learning**. This activity encompasses the production of this thesis' mid-term and final report and reflection on knowledge production.

4. System Architecture Proposal

As was mentioned in subsection 3.1, the application will be subdivided into a graphical editor that allows designers to create Petri Nets to model actor behaviors and choreographies and an interpretation engine that simulates flow of said Petri Nets, translating its semantics into game-engine code. This application is meant to be integrated into an existing game engine (potential candidates are displayed in Table 1 to Table 18).

4.1 Language Functional Specification

The language used in this solution proposal contains a similar syntax to that of Hierarchical Petri Nets (Aalst, 2011) with weighted arcs. By utilizing the capability of grouping sub-nets, this language is able to reduce graphical complexity and thus, improve readability. Another important advantage is that it promotes component reutilization, i.e. the same sub-net can be used in different contexts.

4.1.1 Syntax

This section presents the grammar underlying the language. Although, this is a visual language, spatial attributes of the net were disregarded as they have no effect on the language's syntax. This grammar is presented in the subsequent table:

Table 22 Language Grammar

Terminals	Non Terminals	Production Rules
Places (P)	START	START → (PRE+ MIDDLE POS+)*
Input Places (IP)	PRE	PRE → (PLACE | INPUTPLACE) ARC
Output Places (OP)	MIDDLE	MIDDLE → T | SUB
Fused Places (FP)	POS	POS → ARC (OUTPUTPLACE | PLACE)
Transitions (T)	SUB	SUB → (FUSEDPLACE A MIDDLE A FUSEDPLACE) START
Arc (A)	PRESUB	PLACE → P TK*
Token (TK)	POSSUB	INPUTPLACE → IP TK*
Weight (W)	PLACE	OUTPUTPLACE → OP TK*
	INPUTPLACE	FUSEDPLACE → FP TK*
	OUTPUTPLACE	ARC → A W+
	FUSEDPLACE	
	ARC	

4.1.2 Semantics

A game/simulation model is represented by a root Petri Net that contains set of Petri Nets, each symbolizing a different actor archetype (it is worth noting that instances of an actor archetype share the same Petri Net), as shown in Figure 7.

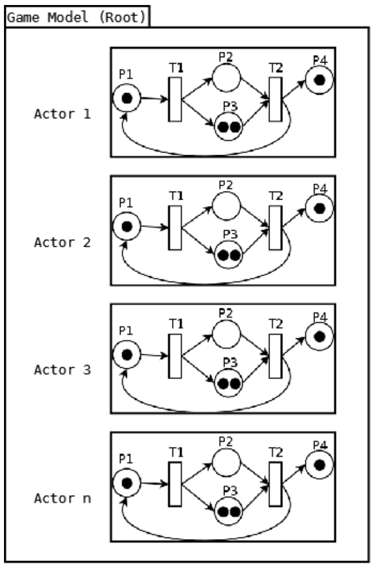


Figure 7 Representation of a Game Model according to the language's specification

Every child of "root" is an independent net that is associated to an actor and follows the language syntax detailed earlier. These nets cannot communicate directly with one another by means of arcs.

Places can be of four types: Input Places, Output Places, Fused Places or Regular Places (this designation must not be confused with the naming given to places linked to/from a transition on the original Petri Net language). Regular Places share the same meaning as places in the Petri Net language. Input Places act as actor sensors. This means that when a token arrives at these special places, something was perceived by the actor. Output Places, on the other hand, assume the role of announcers, i.e. when they receive a token, it is announced to the game world that something has happened. Furthermore, an Input Place can be used to observe an Output Place. Finally, Fused Places are places inside sub-nets that are linked with outer-net places, acting as proxies for their outer-net counterparts. Whenever an outer-net place receives/loses a token, its Fused Place receives/loses the same one as well. Unlike places, which have four different types, Tokens, just like in the Petri Net language, stand for a condition that was met. Transitions, however, have associated programming scripts that govern actions. When a transition fires, its script is executed, meaning that an action is taking place.

4.2 Proposed Design for the Petri Net Editor

The interface's design originated from an iterative process. Initially, a paper prototype was constructed. This prototype was then evaluated through user testing so that it could be refined. After several iterations, the prototype was converted into a mockup representation using Balsamiq, as illustrated in Figure 8.

This proposal is adequate because the interface's viewport provides with the necessary information for the simulation and manipulation of the language's constructs in a segmented way. Because of this, users can easily interact with the editor without having to navigate through menus in order to look for actions. Furthermore, the spatial distribution helps organize the information so that users don't feel overwhelmed.

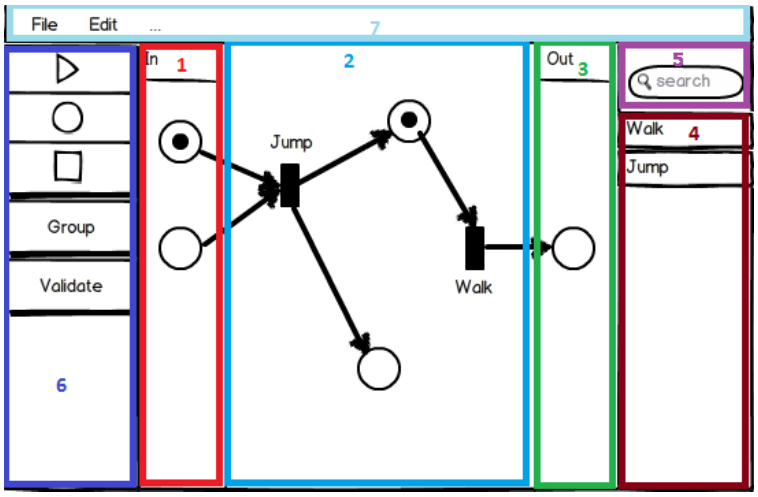


Figure 8 Mockup of the petri net editor's GUI

As can be seen in Figure 8, the editor is divided into 7 panels or menus:

1. **Input Panel**. Here, designers are allowed to put places that represent the actor's inputs. These inputs can range from sensors (vision, hearing, etc…) to messages containing information.
2. **Function Panel**. In this panel, designers can model the actors' logic. As depicted in the figure, this logic can contain both places and transitions as each may have.
3. **Output Panel.** Similar to the Input Panel, places set in this panel represent the actor's outputs, or information he transmits to the game world.
4. **Group Panel.** A panel that lists grouped Petri Nets.
5. **Search Panel.** A panel used to filter the Group Panel, through the means of a keyword search.
6. **Button Sidebar.** A sidebar containing the most important action buttons. From top to bottom, these buttons are: Play, to simulate the given petri net; Add Place, as the name indicates, inserts a place onto one of the panels 1 to 3, as given by the cursor; Add Transition, functioning as Add Place but adding a transition instead; Group, used to gather selected sub Petri Nets into one transition; and Validate to check whether the petri net is valid or not.
7. **Menu Bar.** A menu bar where designers can save/load petri nets to/from disk and import/export groups, if implemented.

4.2.1 Editor Prototype

Loosely inspired by PIPE, this project's editor, is planned to provide the user with the following tasks (tasks are succeeded by the their requirement priority):

- Add places onto the screen (**MUST**)
- Add transitions onto the screen (**MUST**)
- Add tokens in places (**MUST**)
- Remove places from the screen (**MUST**)
- Remove transitions from the screen (**MUST**)
- Remove tokens from places (**MUST**)
- Remove arcs from the screen (**MUST**)
- Link places to transitions and vice-versa through arcs (**MUST**)
- Group Petri Nets (**SHOULD**)
- Edit groups (**SHOULD**)

- Name places, tokens, transitions, arcs or groups (**MUST**)
- Filter groups according to keywords (**NICE**)
- Export groups (**NICE**)
- Import groups (**NICE**)
- Save Petri Nets to disk (**MUST**)
- Load Petri Nets from disk (**MUST**)
- Undo action (**SHOULD**)
- Redo action (**SHOULD**)

4.2.2 Editor Architecture Proposal

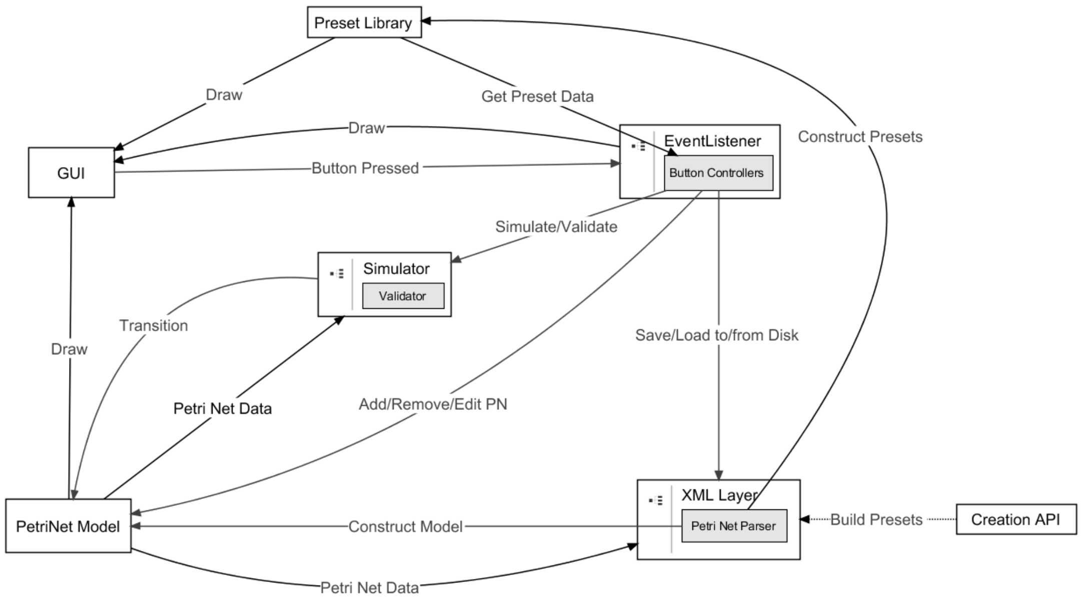


Figure 9 Communication among Editor components

As shown in Figure 9, the editor's inner structure is composed of 7 modules:

- **Petri Net Model**. This module contains the Petri Net's in-memory computational representation and corresponds to the net that the designer is editing at a given moment. It provides the necessary data for other modules such as XML Layer and Simulator to store the net to disk and simulate the net's flow respectively.
- **GUI**. This corresponds to the visual aspect of the editor. It provides a way to render visual components such as Petri Nets, menu bars and buttons and acts as a container for GUI elements. It notifies the Event Listener module when there is a button press event and receives draw calls from the Petri Net Model, Preset Library and the Event Listener.
- **Event Listener**. Module responsible for the buttons' presentation and actions. It contains a controller that communicates with other modules by sending commands in response to received button pressed events.
- **Simulator**. This represents the Petri Nets' simulator and validator. It is used check for Petri Nets' syntax errors and to simulate the net's flow for debugging purposes, so that designers can observe how their creation works.
- **XML Layer**. This layer is accountable for the net's persistence. It contains services to save/load Petri Nets to/from disk, using an XML format. It also contains a parser

used to assemble Petri Nets from XML files so that they can be loaded into models for the Petri Net Model and Preset Library.

- **Preset Library**. This Library contains a memory representation of sub-net presets.
- **Creation API**. Finally, this module provides functions that support the creation of building blocks to populate the Preset Library module. For the time being, this API will be an XSD file. This module works in conjunction with XML Layer to elaborate XML files depicting the referred building blocks.

A more in-depth analysis is provided in Figure 10's class diagram:

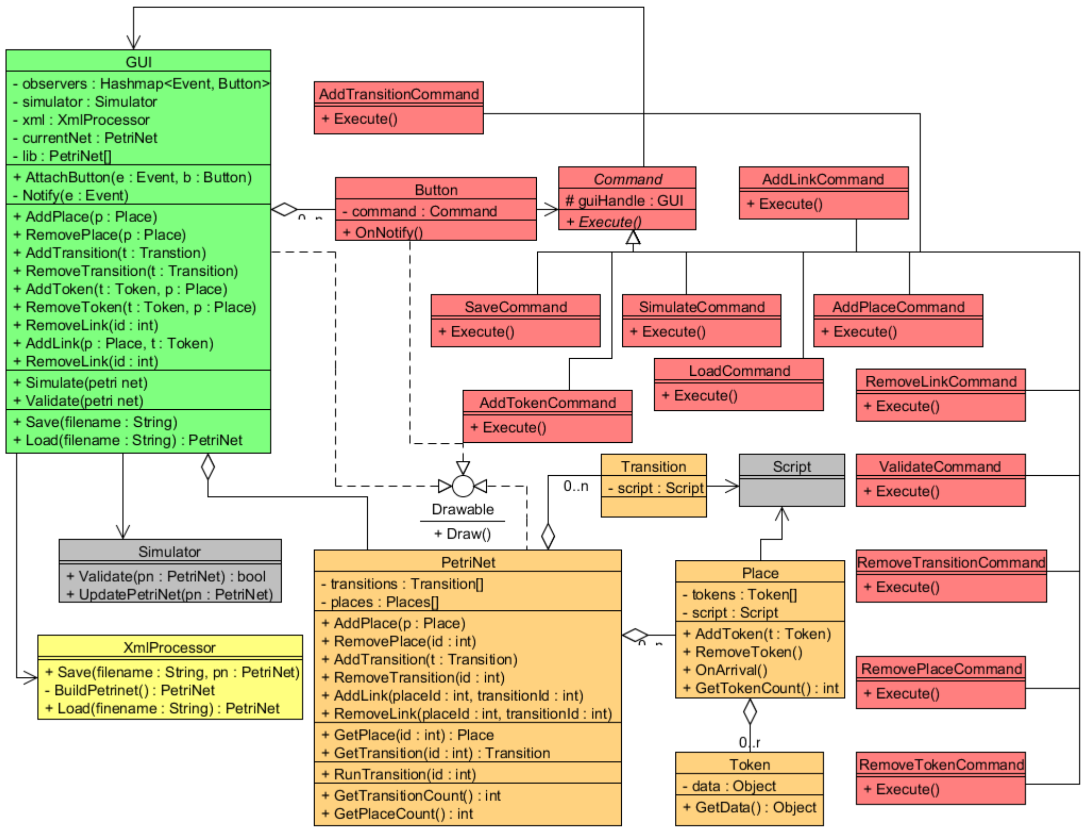


Figure 10 Petri Net Editor Class Diagram. Classes in orange symbolize Petri Nets' data structures; In red, they represent button actions; Green denotes the GUI class; Grey represents the Simulation and execution engine; The XML processing class is depicted in yellow.

The GUI class acts as a Façade for other classes by providing access methods to its component classes. This class communicates events to its collection of Button through a Publish/Subscribe (Observer) (Gamma, Helm, Johnson, & Vlissides, 2011) mechanism. This class contains a Drawable interface which allows it to render itself and its Drawable children (Button and PetriNet).

The Button class acts as a handler for a single Command (Gamma, Helm, Johnson, & Vlissides, 2011). When notified, it executes its command (Figure 10 shows, in red, different commands based on the user task list in 4.2.1).

The groups of classes underlying the Petri Net Model, depicted in orange, are based on the framework presented in Nadle (2003). Each PetriNet object contains a set of Transition and Place, which in turn contains a set of Token. These objects can be identified by an integer.

PetriNet, who is also a Façade, provides methods to add and remove places, transitions, arcs and tokens. The Script class abstracts scripts mentioned in subsection 4.1.

4.3 Petri Net's Simulation and Interpretation Engine

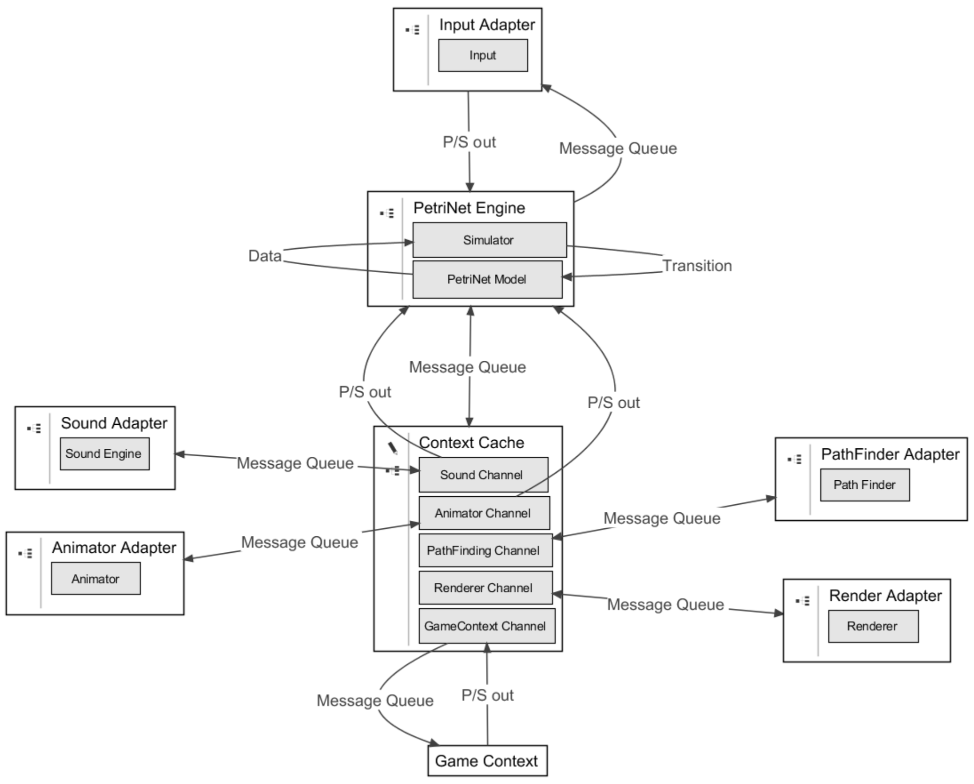


Figure 11 Petri Net Engine Architecture Diagram

Figure 11 exhibits an overview of the system's proposed architecture. As mentioned in 3.1, this system is to be integrated into an existing game engine; hence it is illustrated, in the diagram, components such as Sound Engine, Renderer, Pathfinder, Input and Animator. Regarding the Pathfinder, it is worth noting that this proposal makes the assumption that this module is present (as it is present in most modern game engines).

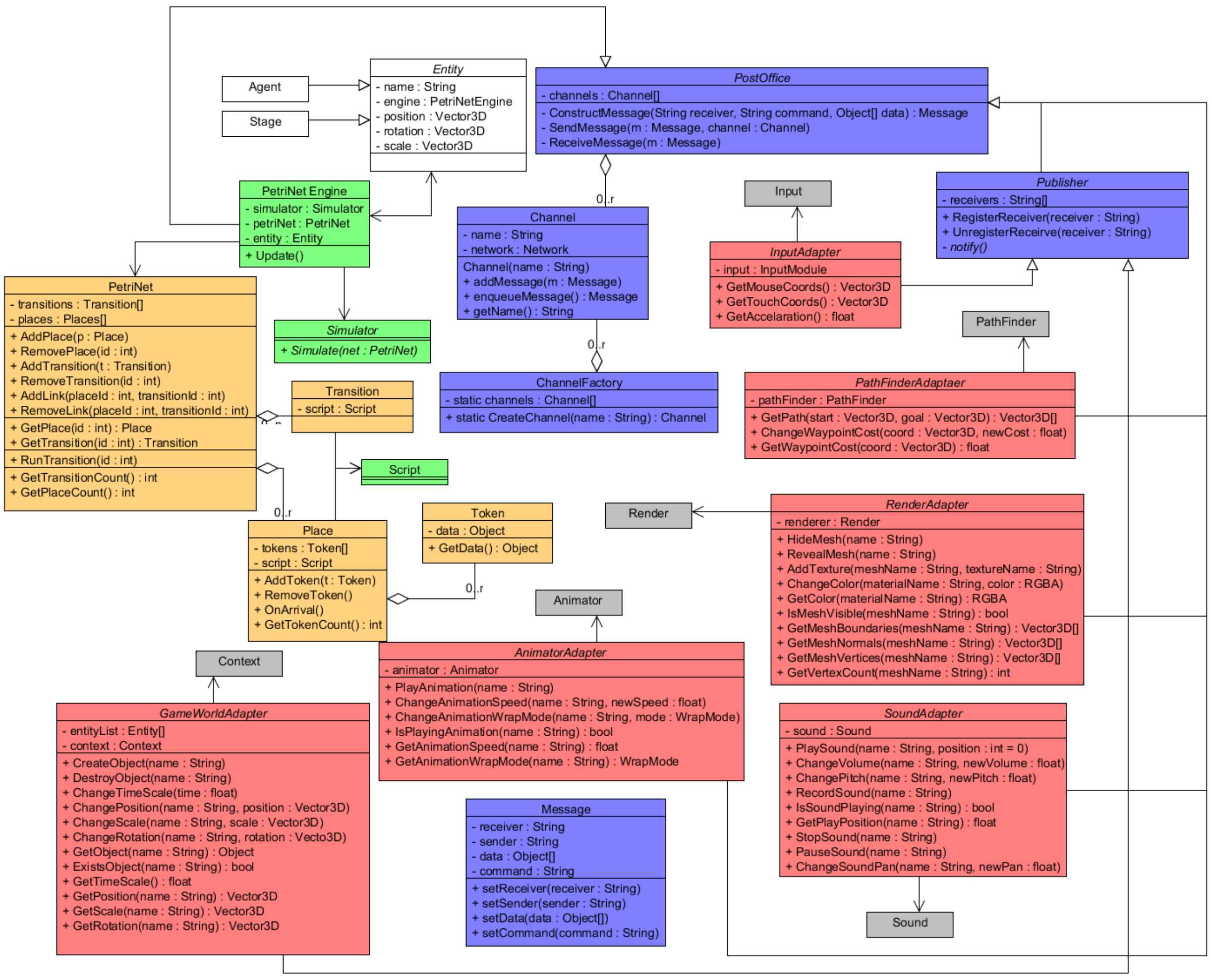
Components exchange information, through messages, using two types of channels: message queues and publish/subscribe channels (referred to as P/S out). Message queues are used to transmit request/response commands among components in a synchronous manner. Publish/Subscribe channels; on the other hand, are used to transmit information asynchronously, as it becomes available, in order to avoid periodic information polling.

The PetriNet Engine is meant to run the actors' Petri Nets, translating them into game-engine code. It contains a Simulator, which supports flow progression in a given Petri Net, and a Petri Net Model module, containing the memory representation of a net.

The Game Context is the collection of actors and other entities (such as scenery objects) that compose the video game. In this proposal, there is a component called Context Cache, which is a mirrored replica of the Game Context. Context Cache also serves as a communication hub between the PetriNet Engine and the other engine components by containing specific channels for each component (because the number of components is static, this allows to avoid unnecessary overhead introduced by message rerouting). This is because most commands also affect the game context, and consequently, its replica.

Figure 12 illustrates the class diagram for the proposed architecture formerly presented: Classes in blue correspond to the message channels and handlers necessary to make the communication function; Classes in red represent the engine's components' (in grey) adapters (Gamma, Helm, Johnson, & Vlissides, 2011); In orange, are the classes comprising the Petri Net Model (as explained in 4.2.2); Green classes represent the PetriNet Engine and its simulator, which uses a strategy pattern (Gamma, Helm, Johnson, & Vlissides, 2011) to allow the integration of different algorithms of Petri Net simulation for testing purposes; Finally, there are a group of classes representing the game objects present in the Game Context and Context Cache (these are Entity, Actor and Stage).

The following subsections explain, in detail, how communication is made between the PetriNet Engine and the rest of the components.



Agent
Stage
Entity
- name : String
- engine : PetriNetEngine
- position : Vector3D
- rotation : Vector3D
- scale : Vector3D
PostOffice
- channels : Channel[]
- ConstructMessage(String receiver, String command, Object[] data) : Message
- SendMessage(m : Message, channel : Channel)
- ReceiveMessage(m : Message)
PetriNet Engine
- simulator : Simulator
- petriNet : PetriNet
- entity : Entity
+ Update()
Channel
- name : String
- network : Network
Channel(name : String)
+ addMessage(m : Message)
+ enqueueMessage() : Message
+ getName() : String
0..r
Input
InputAdapter
- input : InputModule
+ GetMouseCoords() : Vector3D
+ GetTouchCoords() : Vector3D
+ GetAccelaration() : float
Publisher
- receivers : String[]
+ RegisterReceiver(receiver : String)
+ UnregisterReceive(receiver : String)
- notify()
PetriNet
- transitions : Transition[]
- places : Places[]
+ AddPlace(p : Place)
+ RemovePlace(id : int)
+ AddTransition(t : Transition)
+ RemoveTransition(id : int)
+ AddLink(placeId : int, transitionId : int)
+ RemoveLink(placeId : int, transitionId : int)
+ GetPlace(id : int) : Place
+ GetTransition(id : int) : Transition
+ RunTransition(id : int)
+ GetTransitionCount() : int
+ GetPlaceCount() : int
Simulator
+ Simulate(net : PetriNet)
PathFinder
ChannelFactory
- static channels : Channel[]
+ static CreateChannel(name : String) : Channel
PathFinderAdaptaer
- pathFinder : PathFinder
+ GetPath(start : Vector3D, goal : Vector3D) : Vector3D[]
+ ChangeWaypointCost(coord : Vector3D, newCost : float)
+ GetWaypointCost(coord : Vector3D) : float
Transition
- script : Script
Script
Token
- data : Object
+ GetData() : Object
Render
RenderAdapter
- renderer : Render
+ HideMesh(name : String)
+ RevealMesh(name : String)
+ AddTexture(meshName : String, textureName : String)
+ ChangeColor(materialName : String, color : RGBA)
+ GetColor(materialName : String) : RGBA
+ IsMeshVisible(meshName : String) : bool
+ GetMeshBoundaries(meshName : String) : Vector3D[]
+ GetMeshNormals(meshName : String) : Vector3D[]
+ GetMeshVertices(meshName : String) : Vector3D[]
+ GetVertexCount(meshName : String) : int
Place
- tokens : Token[]
- script : Script
+ AddToken(t : Token)
+ RemoveToken()
+ OnArrival()
+ GetTokenCount() : int
Animator
Context
AnimatorAdapter
- animator : Animator
+ PlayAnimation(name : String)
+ ChangeAnimationSpeed(name : String, newSpeed : float)
+ ChangeAnimationWrapMode(name : String, mode : WrapMode)
+ IsPlayingAnimation(name : String) : bool
+ GetAnimationSpeed(name : String) : float
+ GetAnimationWrapMode(name : String) : WrapMode
GameWorldAdapter
- entityList : Entity[]
- context : Context
+ CreateObject(name : String)
+ DestroyObject(name : String)
+ ChangeTimeScale(time : float)
+ ChangePosition(name : String, position : Vector3D)
+ ChangeScale(name : String, scale : Vector3D)
+ ChangeRotation(name : String, rotation : Vecto3D)
+ GetObject(name : String) : Object
+ ExistsObject(name : String) : bool
+ GetTimeScale() : float
+ GetPosition(name : String) : Vector3D
+ GetScale(name : String) : Vector3D
+ GetRotation(name : String) : Vector3D
Message
- receiver : String
- sender : String
- data : Object[]
- command : String
+ setReceiver(receiver : String)
+ setSender(sender : String)
+ setData(data : Object[])
+ setCommand(command : String)
SoundAdapter
- sound : Sound
+ PlaySound(name : String, position : int = 0)
+ ChangeVolume(name : String, newVolume : float)
+ ChangePitch(name : String, newPitch : float)
+ RecordSound(name : String)
+ IsSoundPlaying(name : String) : bool
+ GetPlayPosition(name : String) : float
+ StopSound(name : String)
+ PauseSound(name : String)
+ ChangeSoundPan(name : String, newPan : float)
Sound

Figure 12 Petri Net Engine Class Diagram

4.3.1 Animator Module

The Animator component is used to play animations, as such, it provides services to play or stop them, to change its parameters and to inform that a certain animation is playing, as seen in Figure 13. Command messages labeled as (1), originated from the PetriNet Engine, are relayed through the Context Cache to instruct the animator module. It then responds sending the results of (1) commands back to the PetriNet Engine, through the Context

Cache. When available, the Animator sends information depicting (2) to the Context Cache to notify the PetriNet Engine.

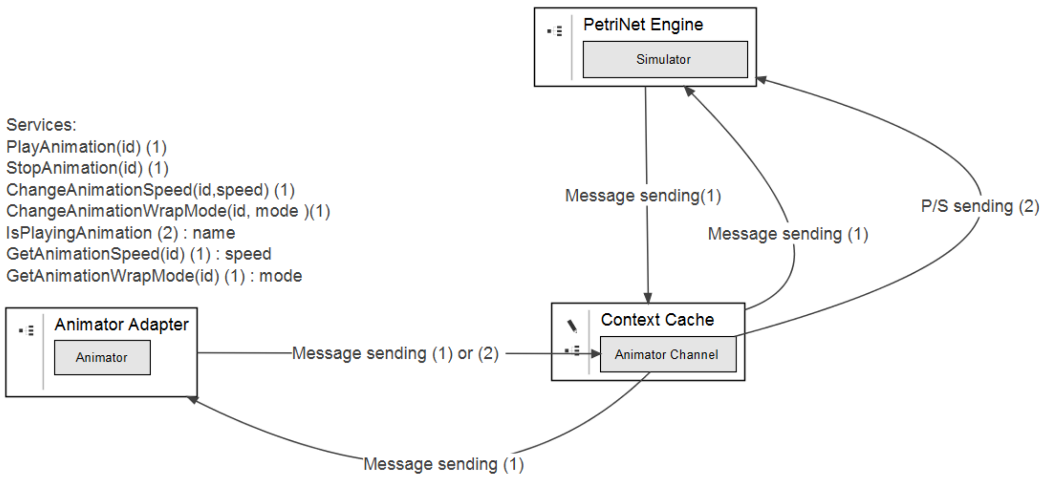


Figure 13 Communication between Animator and Petri Net Engine

4.3.2 Game Context Module

The Context Cache, Figure 14, provides services to retrieve and manipulate game objects (such as actors). PetriNet Engine makes (1) and (2) requests. Since (2) requests result in an inconsistency between the Game Context and the Context Cache, these type of requests originate a Game Context update. Game Context also sends updates to Context Cache, through a publish/subscribe channel.

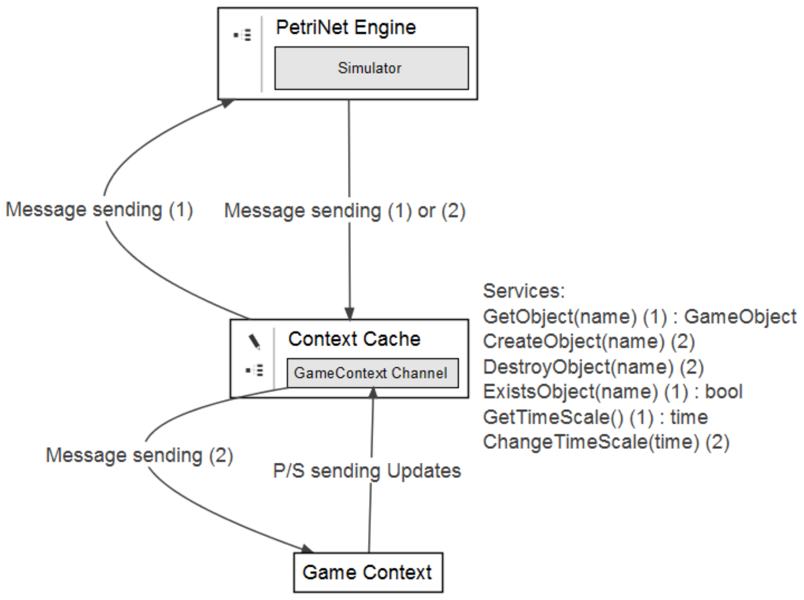


Figure 14 Communication between Game Context and Petri Net Engine

4.3.3 Input Module

This component, shown in Figure 15, is used to process the input (such as mouse, keyboard, touchpads, gamepads, etc…). Data labeled as (1) is sent directly to the PetriNet Engine, through a publish/subscribe channel, since its results do not directly affect the game context. The PetriNet Engine sends requests labeled as (2) to the Input component which are responded through regular messages.

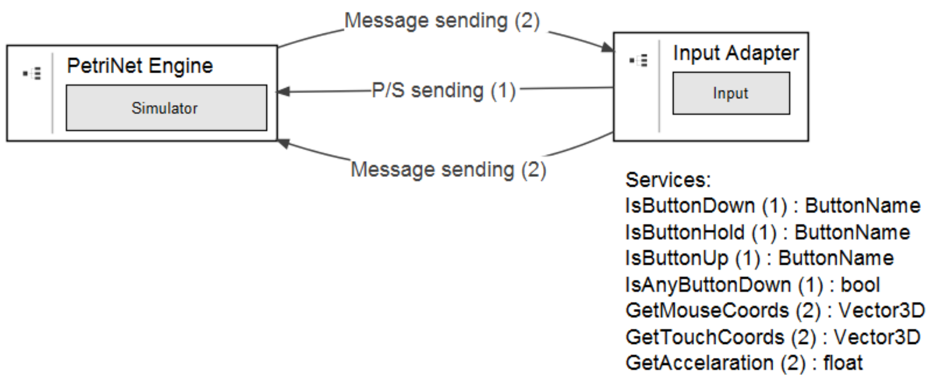

Message sending (2)
PetriNet Engine
Simulator
P/S sending (1)
Input Adapter
Input
Message sending (2)
Services:
IsButtonDown (1) : ButtonName
IsButtonHold (1) : ButtonName
IsButtonUp (1) : ButtonName
IsAnyButtonDown (1) : bool
GetMouseCoords (2) : Vector3D
GetTouchCoords (2) : Vector3D
GetAccelaration (2) : float

Figure 15 Communication between Input and Petri Net Engine

4.3.4 Pathfinder Module

Figure 16 illustrates this component, whose purpose is to calculate paths in a given map. The PetriNet Engine sends requests labeled as (1) or (2). Since only (1) commands require response, (2) commands prompt a change in parameters used to calculate a path, these responses are sent, from the Pathfinder, back to the PetriNet Engine.

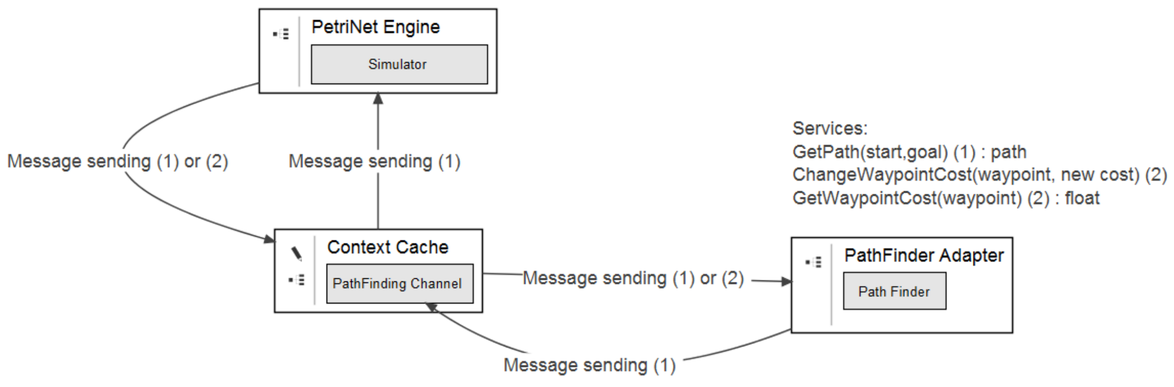

PetriNet Engine
Simulator
Message sending (1) or (2)
Message sending (1)
Services:
GetPath(start,goal) (1) : path
ChangeWaypointCost(waypoint, new cost) (2)
GetWaypointCost(waypoint) (2) : float
Context Cache
PathFinding Channel
Message sending (1) or (2)
PathFinder Adapter
Path Finder
Message sending (1)

Figure 16 Communication between Pathfinder and Petri Net Engine

4.3.5 Render Module

This module, in Figure 17, refers to how the visual feedback is presented and manipulated. As such, services provided by the Context Cache that manipulate objects' position, rotation and scale, are presented in this diagram. PetriNet Engine sends (1) and (2) requests either directly to the Context Cache or to the Renderer (relayed through the Context Cache). Responses are sent back to the PetriNet Engine.

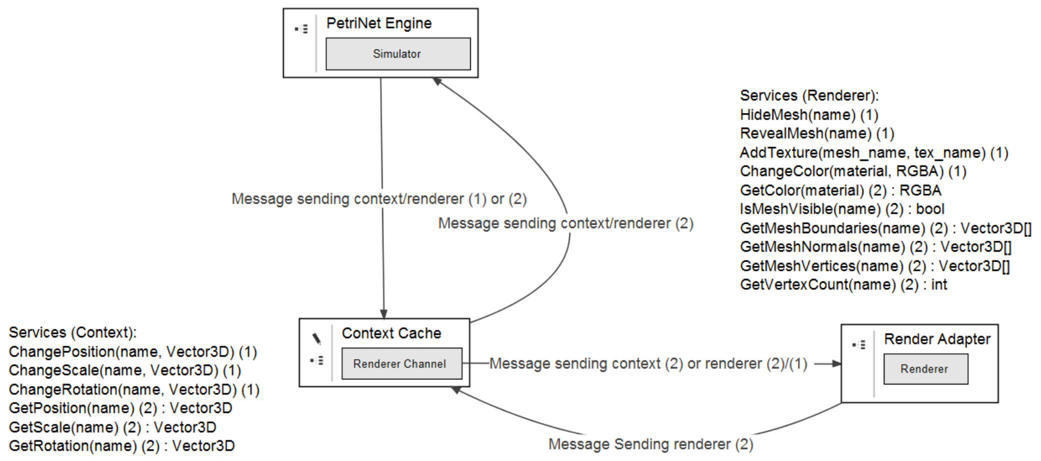

PetriNet Engine
Simulator
Services (Renderer):
HideMesh(name) (1)
RevealMesh(name) (1)
AddTexture(mesh_name, tex_name) (1)
ChangeColor(material, RGBA) (1)
GetColor(material) (2) : RGBA
IsMeshVisible(name) (2) : bool
GetMeshBoundaries(name) (2) : Vector3D[]
GetMeshNormals(name) (2) : Vector3D[]
GetMeshVertices(name) (2) : Vector3D[]
GetVertexCount(name) (2) : int
Message sending context/renderer (1) or (2)
Message sending context/renderer (2)
Services (Context):
ChangePosition(name, Vector3D) (1)
ChangeScale(name, Vector3D) (1)
ChangeRotation(name, Vector3D) (1)
GetPosition(name) (2) : Vector3D
GetScale(name) (2) : Vector3D
GetRotation(name) (2) : Vector3D
Context Cache
Renderer Channel
Message sending context (2) or renderer (2)/(1)
Render Adapter
Renderer
Message Sending renderer (2)

Figure 17 Communication between Renderer and Petri Net Engine

4.3.6 Sound Engine Module

The Sound Engine is used to play and manipulate sounds. As depicted in Figure 18, it provides services to record sounds, to play, stop or pause sounds, to change sound characteristics such as its pitch, volume or pan value. PetriNet Engine sends (1) and (3) requests, from which only (3) generate responses from the Sound Engine. The Sound Engine also sends data labeled as (2) so that the Context Cache can notify the PetriNet Engine.

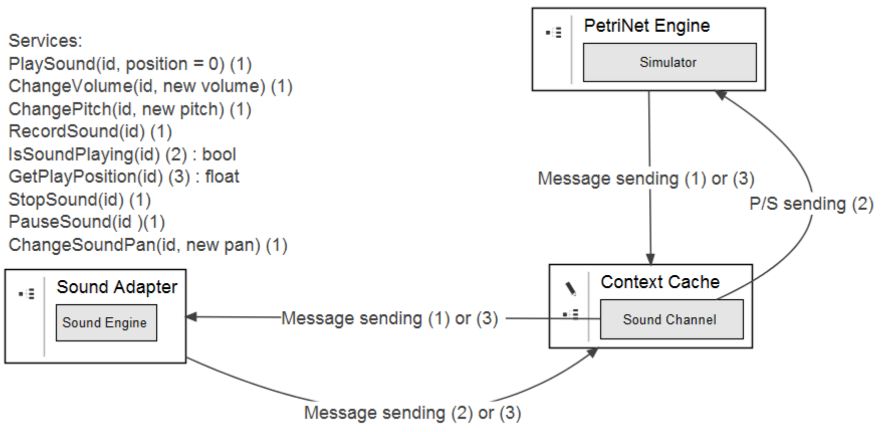

Services:
PlaySound(id, position = 0) (1)
ChangeVolume(id, new volume) (1)
ChangePitch(id, new pitch) (1)
RecordSound(id) (1)
IsSoundPlaying(id) (2) : bool
GetPlayPosition(id) (3) : float
StopSound(id) (1)
PauseSound(id)(1)
ChangeSoundPan(id, new pan) (1)
PetriNet Engine
Simulator
Message sending (1) or (3)
P/S sending (2)
Sound Adapter
Sound Engine
Message sending (1) or (3)
Context Cache
Sound Channel
Message sending (2) or (3)

Figure 18 Communication between Sound and Petri Net Engine

4.4 Features

With this system, it is intended to provide the following features:

- Visual debugging of the Petri Net through simulation.
- Reuse of grouped Petri Nets.
- Develop preset Petri Nets.
- A means to create scripts and associate them with transitions.
- Integration with arbitrary game/simulation engines.
- Support for multiplayer games.
- Support for multiple actor modeling.

4.5 Quality Attributes

Intrinsically, this tool was designed to provide the ensuing quality attributes: portability so that it would not be tied to a particular OS, interoperability which would allow to use the tool with different game/simulation engines, usability to complement Petri Net's accessibility, error recovery due to the fact that designers will be working with a language with a defined syntax and, therefore, must be warned of syntax errors and how to solve them and, finally, scalability relative to the number of actors.

The proposed architecture fulfills all but two quality attributes: it provides adapters so that various game/simulation engines can be used to integrate the application, thus satisfying interoperability; the architecture includes a syntax validator module so that it can be used to issue warning messages, ensuring error recovery; it is composed of a local game context cache to allow local computations instead of filling the bandwidth and, consequently, the game server, therefore promoting scalability. Portability and usability are not fulfilled by the system's architecture because, the former is dependent on the technology used and the latter relates to the application's interface.

5. Implementation

5.1 System Overview

An overview of the entire system, and how it communicates with Unity, is presented in Figure 19. As can be seen, the editing application contains three execution threads: GUI, Execution Engine and Communication Module. On the Unity-side, there are two execution threads: The game logic thread, or main thread, which is responsible to handle all of the games' computations and the Communication Module. Both Communication Modules exchange information with each other.

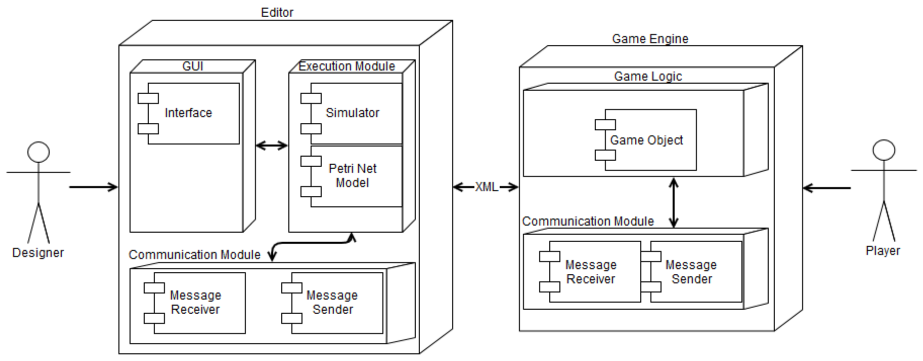


Figure 19 Deployment Diagram of the Architecture

A more in-depth view of the editor is presented in Figure 20.

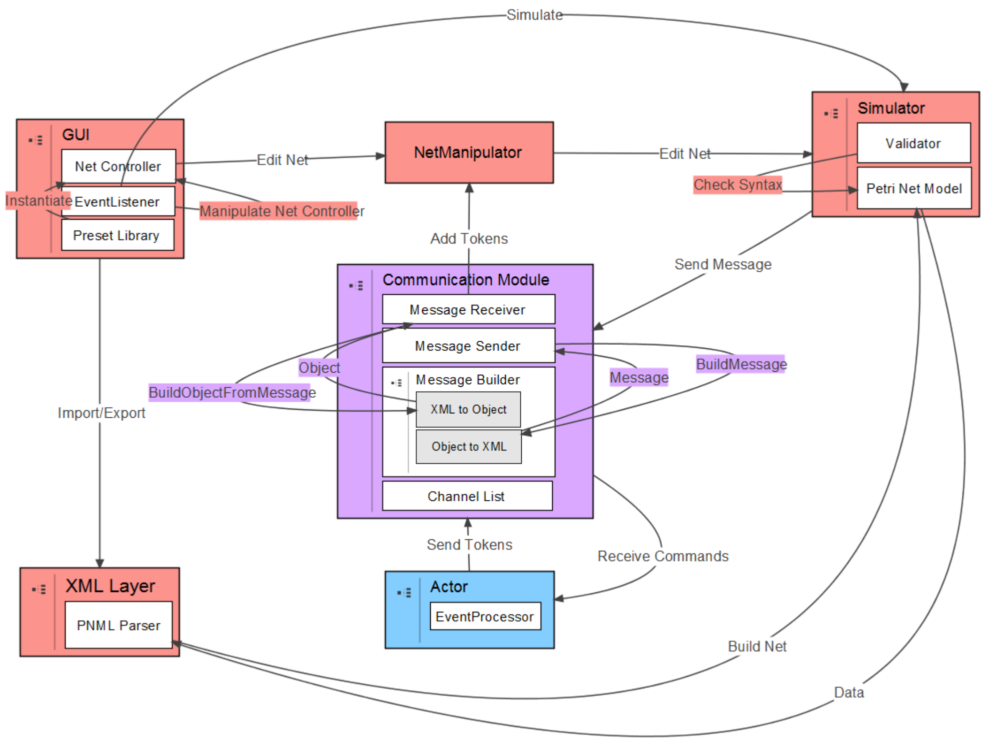


Figure 20 Static Perspective on the Architecture of the Application. Red boxes indicate editor's components while the blue box symbolizes a Unity module. Finally, the purple box represents a component that both the editor and Unity contain.

As illustrated, the editor contains 6 components while the game engine contains 2:

- **GUI** – This block contains elements that make for the interface including EventListener which is a module that handles input events. It also contains Net Controllers which are objects composed of a single Net element (transitions, places or arcs) and its respective view. These controllers encompass a set of rules that indicate which interactions are available for a given element and, likewise, how the net's view is presented in the GUI's viewport; The PresetLibrary which is a container that stores presets of sub nets so that they can be reused later.
- **XML Layer** – This component, comprised of PNML Parser, converts a given Petri Net model to PNML and vice-versa.
- **Net Manipulator** – This module serves as a Façade (Gamma, Helm, Johnson, & Vlissides, 2011) with a thread safe interface that allows manipulating the Petri Net model. This is due to the fact that the Execution Engine, which contains the Petri Net data structure, and the GUI run on separate threads.
- **Simulator** – The simulator performs simulation steps in order to fire transitions as well as to trigger user-defined actions. This component contains the Petri Net data structure as well as its syntax validator.
- **Communication Module** – This is the bridge between the game engine and the editor. It provides services to send and receive XML messages. As such, the Communication Module also includes an XML parser and an object builder. It also contains a list of observers (Gamma, Helm, Johnson, & Vlissides, 2011) used to notify special Petri Net places (See subsection 4.1.2) that a token is available and on the Unity side that a command has been received.
- **Actor** – Represents an in-game actor. Each actor contains an Event Processor component. This component receives instructions from the editor, relayed through the Communication Module and, likewise, sends sensor perceived data through the same model as well.

Communication, as hinted, is done through the exchange of XML messages much like the SOAP protocol. But unlike SOAP, the messages' syntax is lighter as to avoid an unnecessary decrease in performance due to message processing. Seeing that one of the attribute qualities of the system is interoperability, an XSD defining the syntax was created with the purpose of being publicly accessible, so that game engines may use it to communicate with the editor. The next code snippet illustrates the XSD file's contents.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
         targetNamespace="MessageSyntax"
   elementFormDefault="qualified" xmlns:tns="MessageSyntax">
   <xsd:element name="Message">
      <xsd:complexType>
         <xsd:choice>
            <xsd:element name="Body">
               <xsd:complexType>
                  <xsd:attribute name="type" type="xsd:string"/>
               </xsd:complexType>
            </xsd:element>
         </xsd:choice>
         <xsd:attribute name="sender" type="xsd:string"/>
         <xsd:attribute name="receiver" type="xsd:string"/>
      </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

Figure 21 Messages' XML syntax in XSD

The message consists of a single element with the attributes "sender" and "receiver" that represent the message's sender and receiver respectively. This element contains a Body type child representing data that is being transmitted. Data can be any object as long as it is serialized through XML and its XSD file, detailing an xml-to-object mapping, is shared between the editor and the game engine. However, at the moment of writing, this feature is restricted only to support object wrappers for native types (string, integer, float and boolean) and a special object that contains methods to be invoked on a particular class. The type attribute of the Body element is used to identify the object's type so that it can be de-serialized.

Regarding connections, the communication module contains a pool of worker threads that are meant to handle incoming requests for more than one client. This way, designers can use only one editor which facilitates debugging and run-time authoring of the net (having multiple editors would require that changes made in one net to propagate across the other editors, increasing bandwidth usage and the risk of having inconsistent models). This design choice makes the assumption that game state consistency (how in-game actions preserve partial order among all clients) is the engine's responsibility. This is because state consistency is provided by network middleware and, therefore, it is not necessary to replicate efforts towards the editor system. This way, the execution engine of the system can use more processing power.

The subsequent two figures showcase how the editor works with multiplayer games in peer-to-peer and client/server architectures. Since hybrid architectures mixes communications both peers and servers, an image of this architecture was not included as the interactions are similar to the ones present in the following figures:

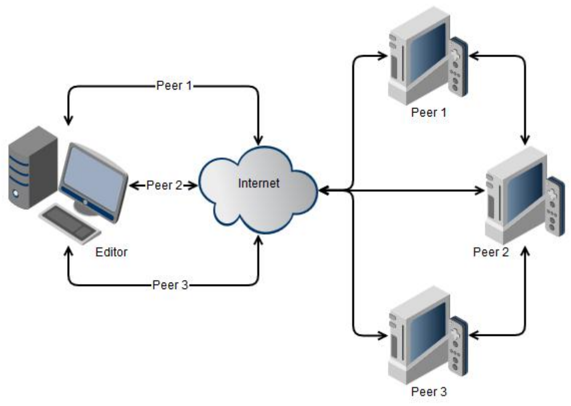


Figure 22 The system in a P2P environment

As can be observed, each peer communicates with every other peer, alongside the editor.

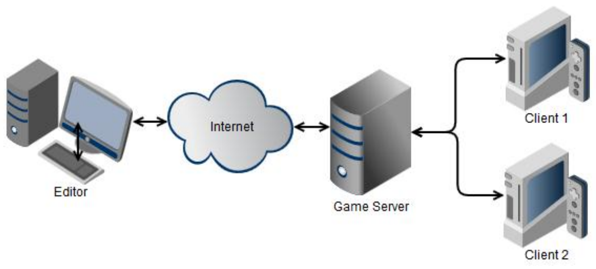


Figure 23 The system in a client/server environment

Unlike the previous picture, here a server acts as a gateway for clients to communicate with the editor.

5.1.1 Petri Net Interpretation Overview

This subsection provides insights, through sequence diagrams, regarding the implemented functionalities that contribute to the underlying mechanism that translates Petri Nets' behavior models into in-game actions. These functionalities include system startup, how tokens are received by input places, how transitions invoke actions and how the editor displays to the user, available channels and scripts. It is worth stating that some classes and methods were omitted from the sequence diagrams as to increase readability and also that the Unity class present in most of the diagrams represents arbitrary objects in the Unity game-engine.

The startup, presented in Figure 24, demonstrates how both the editor and the game engine start and initiate communication. As can be observed, the main editor class, PetriNetEditor, creates a root Net and a first child net, both of type BasicNet. It then creates the net's visual representation, EditorViewport, the simulator and the PostOffice, a class responsible to send and receive network messages. On the Unity-side, the engine creates its own PostOffice which issues a connection request to the editor. This sequence assumes that the editor has been started before Unity.

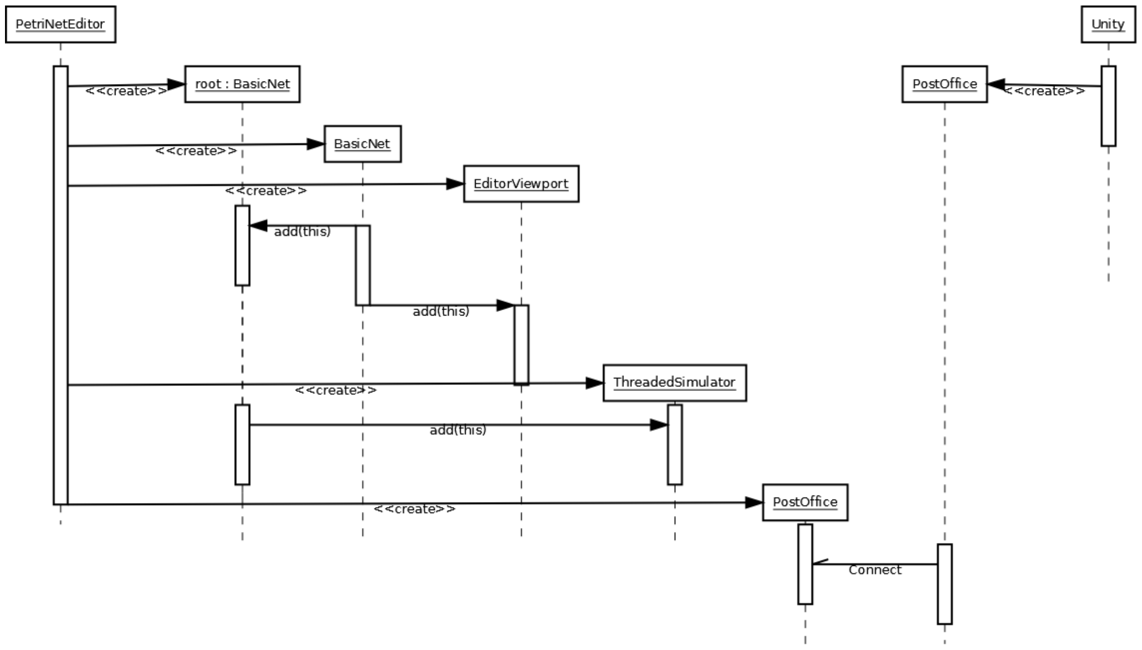


Figure 24 System startup sequence diagram

Input Places can get tokens from two means: they either receive it from the actor's sensors or from an Output Place. In both cases, this is done by sending messages through the PostOffice class even though Input Places and Output Places reside on the same editor. This design choice was made in order to keep the code consistent and to allow for the system to support multiple editors in the future.

Since Petri Nets may represent one or more instances of an actor archetype, knowing which actor sent a token to a given Input Place to cause a transition to fire became a problem. To counter this, tokens where given owners. They can be either owned by a specific actor in which case they carry its id or they can be neutral. Id-bound tokens can only be used to fire transitions if all of the places, linked to that transition, contain tokens with the same id or neutral tokens.

Figure 25 illustrates how an actor in Unity, after perceiving some state from the environment, causes a token to be inserted in an Input Place. On the right-most corner, the actor's component, EventProcessor, starts by building an XML representation of the message (including the state it perceived) using the helper class, MessageBuilder. Afterwards, it uses PostOffice to relay the message to the editor's PostOffice. When the editor's PostOffice receives the message, it unwraps the message's receiver so that it can notify the appropriate Channel observers which are being observed by NetworkInputPlace objects. When NetworkInputPlace objects are issued an update, they retrieve the state from the message and use ThreadSafeNetManipulator to add a token to the affected Input Place.

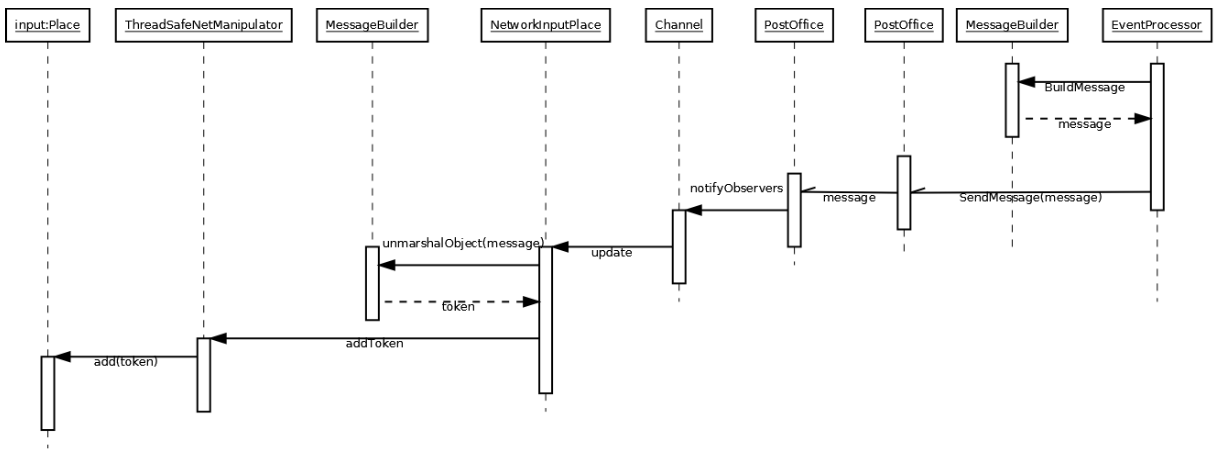


Figure 25 Sending a token from Unity to Input Place sequence diagram

The other way for Input Places to get tokens is described in Figure 26. When an Output Place receives a token from a transition, it notifies a NetworkOutputPlace object. Upon receiving this notification, NetworkOutputPlace objects, with the help of MessageBuilder, create a message with the token and send it through the PostOffice. Afterwards, the algorithm works in the same manner as explained previously.

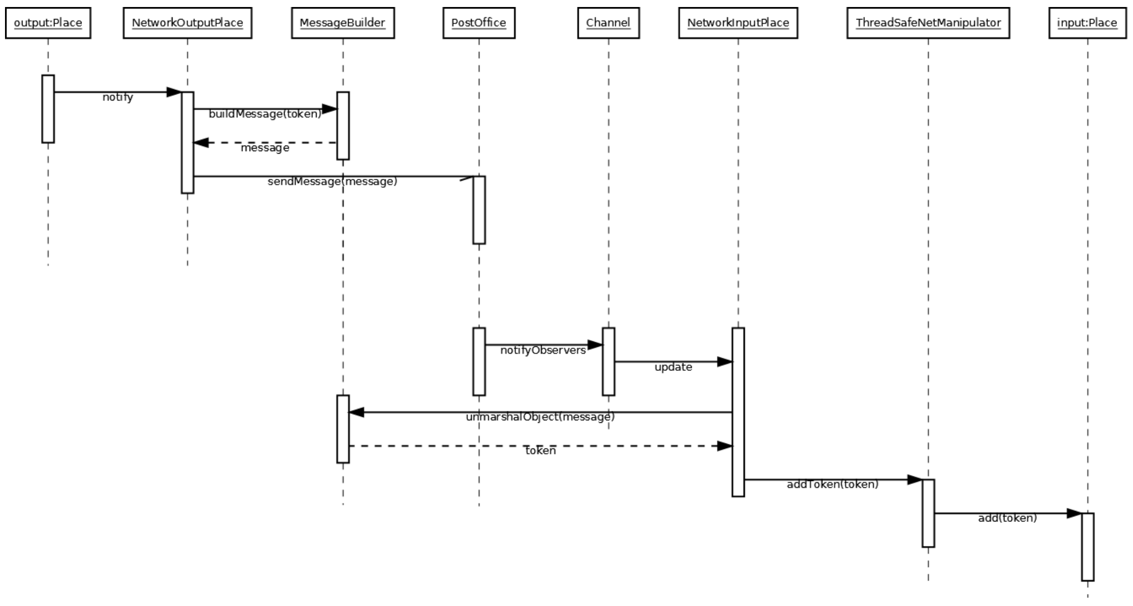


Figure 26 Sending a token from Output Place to Input Place sequence diagram

The following figure showcases how transition firing is converted to actions in-game. Since using technologies akin to RPC had the disadvantage to be non-interoperable, invocation of game engine-side methods is done through reflection (Buschmann, Meunier, Rohnert, Sommerlad, & Stal, 2011). Whenever a transition, containing an action, is fired, a message, detailing which method to invoke, is sent through the PostOffice. However, before sending the message, it is first built using MessageBuilder. After receiving the message in the game engine's PostOffice, it is then relayed to the target actor's EventProcessor which extracts the method to invoke. At this point, one of two situations may occur, depending on the game being online-based or not: if it is not online-based, the method is invoked using the component's SendMessage method. If it is online-based, it is used an RPC invocation (Unity's only method of sending information in a multiplayer game) in order to propagate the state throughout other Unity clients connected to the game server as Unity's multiplayer middleware works by using RPC.

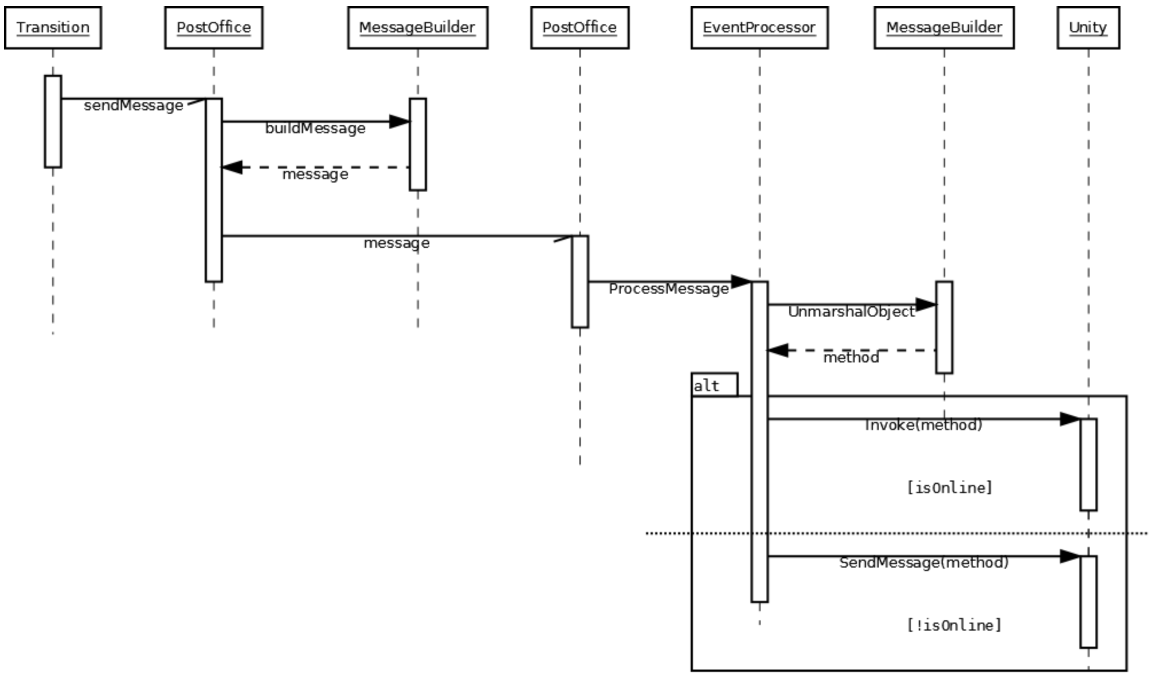


Figure 27 Invoking a script in Unity sequence diagram

Finally, Figure 28, illustrates how the editor gets a list of available scripts and channels to display to the user. Whenever one of these parameters is to be sent, Unity builds a message with the given channel/script name and sends it through the PostOffice. After sending the message to the editor's PostOffice, it is then converted to the channel/script and added to the ApplicationContext, a Façade class that contains the context of the editor's GUI. On a side-note, scripts are confined inside methods that have a special attribute "[PNTransitionMethod]". This way, only methods with this attribute can be sent to the editor in order to avoid cluttering the editor's lists with unusable methods.

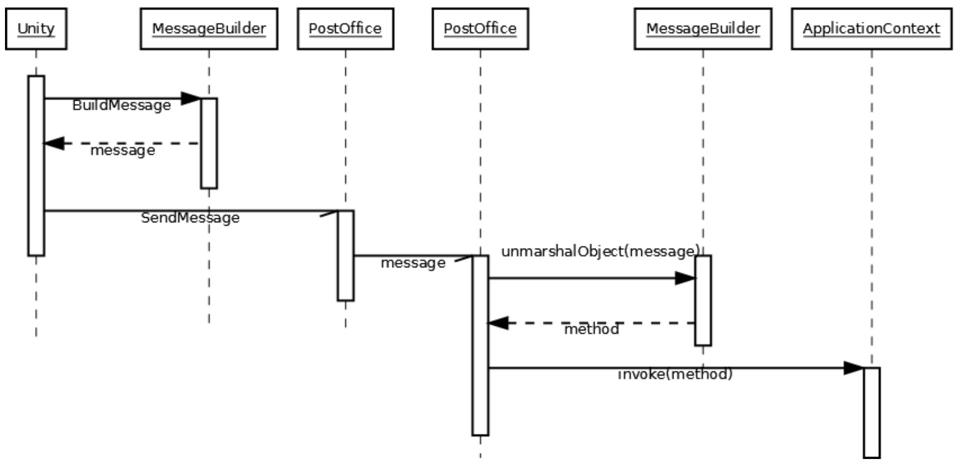


Figure 28 Update the editor with available channels/scripts sequence diagram

5.1.2 Notable Differences from the Original Architecture Proposal

During development, some modifications to the architecture proposal had to be done either due to restrictions imposed by the technology choices or due to some design considerations made during the implementation stage.

The first big difference is that the execution engine and the editor are not separate entities as can be observed by the inclusion of a communication module in the editor's architecture. Initially it was thought to have the editor to create a static model which would then be imported by the execution engine on the game engine's side. Later on, this was found not to be an ideal solution because it would require additional effort to permit interoperability.

In the editor, the creation API module is absent because it was considered deprecated due to the possibility of creating presets using the editor instead. Additionally, the Petri Net Model is inserted in the simulator unlike what was stated in the original proposal. This is because JFern's simulator requires for the model to be part of it. Finally, in the implementation's architecture, there is a module NetManipulator used for thread-safe operations. This was included since the implementation contained more than one thread (an aspect which was not considered in the initial proposal).

In the game engine-side, the differences are more notorious. As can be observed in Figure 20, there are no wrappers for game engine components. Since Unity has a component-driven architecture in which game objects are a collection of behavior components, each containing several engine elements (for example, input, sound or animation), introducing wrappers for such elements would result in rewriting the engine. Instead, a better solution was devised involving the creation of a component that communicated directly with the communication module and offered services to either send messages to other components in the same game object or to send data to the editor through the communication module.

5.1.3 Proof of Concept

A prototype game was developed using the system to showcase its functionalities. In order to focus on the spawning and interaction of multiple autonomous actors I developed a game called Orphibs, drawing inspiration from SEGA's Sonic Adventure's (Sega, 2001) meta game: Chaos Garden. Chaos Garden is an Artificial Life, or A-Life, meta-game in which players control an avatar whose purpose is to take care of autonomous fragile creatures called Chaos. Much like Chaos Garden, Orphibs contains autonomous creatures. But unlike the former, Orphibs does not require player interaction. Another specificity about this game is that it can run on several computers – each spawning its own orphib into the game. This design choice was due to the fact that it facilitates testing in a self-sufficient manner as it doesn't require other users to try out the game. An in-game screenshot can be seen in Figure 29:

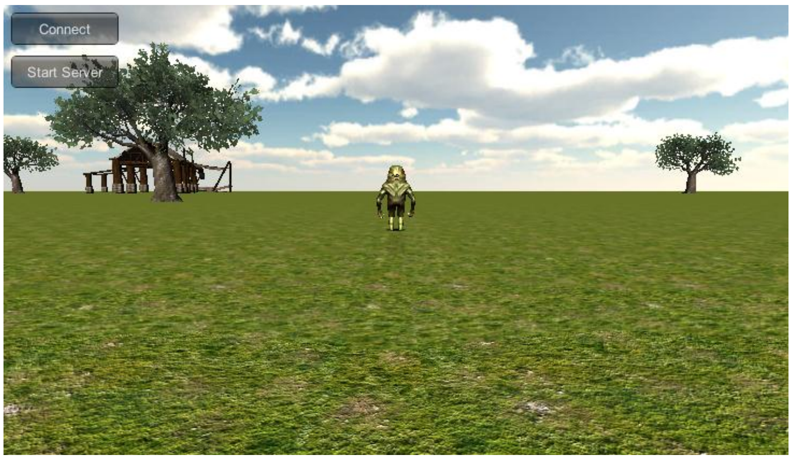


Figure 29 Orphibs screenshot

The game's world consists in a large grass field with several trees, a pond and a wooden house. Occasionally, toy trucks and vegetables are spawned in random locations. Orphibs are small, lime-green and anthropomorphic creatures. Internally, they contain a set of status variables: hunger, boredom, tiredness and age and a list of possible atomic actions: grow, eat, play, sleep, walk, run and stop. Status variables change as time passes in relation to different constants. Actions, on the other hand, are a succession of tasks that span across a fixed time-length and influence how internal status variables, excluding age, are altered. An orphib cannot do two simultaneous actions. An exception to this rule is growing which occurs at every time frame. The remaining actions are chosen according to a Goal-Oriented selection method (Milligton & Funge, 2009) using the internal status variables as utility values.

The editor system was used to model the updating schedule of the internal status, the growing routine and which action took place given an arbitrary selection. This is illustrated in the next figure:

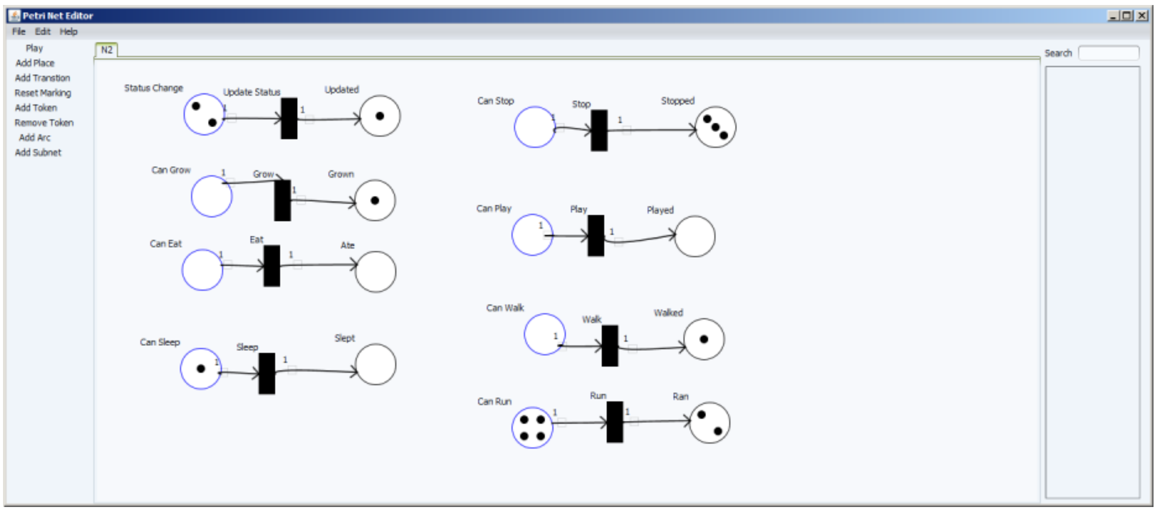


Figure 30 Orphibs' behavior model during the simulation's execution

This model is composed of input places that represent sensors of the orphibs' internal state. For every time frame, the "status change" and "can grow" places are filled with tokens, one for each orphib present in the game. The same thing happens for every other Input Place when an action selection happens. The transitions correspond to scripts that either: update the status variables, make an orphib grow or puts in motion the set of tasks that compose eat, sleep, stop, play, eat, walk or run.

5.2 Technology Choices

The artifact was built using Java due to its portability and the interface was made with Java's GUI API: Swing. Initially, the editor was meant to be based on the JFern Editor because it provided JFern's Petri Net threaded simulation mechanism, data structure and PNML exporting/importing API as well as views and controllers for their visual representation. This idea was discarded because the GUI's code was poorly documented, confusing and some of the it classes were not made available as source code. Consequently, only the simulator, data structure and PNML parser were used. The reason behind using JFern is that besides offering the previously mentioned set tools, it was the only tool from the ones researched in subsection 2.5.2 that allowed to introduce code to be executed when a transition fires, thus reducing some programming effort when developing the editor's execution mechanism. Nevertheless, this tool was not used in its vanilla form but, instead, it was modified to provide some attributes to the Petri Net object's including message channels.

The communication system uses TCP as its transport protocol due to its reliability. Although UDP would seem to be a better option for a transport protocol to ensure network scalability, it would require an application-layer reliability mechanism to guarantee that messages sent from either the game engine or the editor reach their destinations as losing too much datagrams would have a negative impact on games/simulations. Since TCP had a built-in reliability mechanism, this protocol was chosen during the proof of concept to illustrate how the communication works but was implemented in a way that it could be easily substituted by UDP for network scalability tests.

Data exchanged across the network is done by wrapping objects in a XML layer because it is a human-readable format (which facilitates debugging), the de-facto standard in interoperable information exchange protocols and can be natively processed in a variety of programming languages.

The game engine used to showcase the proof of concept was Unity. As stated in subsection 2.3, Unity is a general purpose engine, comprised of several visual tools, with support for several scripting languages, including C# .NET. The reason behind this choice was that its visual tools simplified the process of developing a proof of concept prototype whilst the native support for C# .NET allowed for an easier integration with the editor as the functionalities present in the .NET framework could be used. This allowed to circumvent Unity's network limitation: only Unity games can communicate with each other through the build-in middleware. In fact, Unity-side communication was made using TCP connections as well.

Unity is a single-threaded engine. This means that blocking operations, for instance, waiting for connections could disrupt the game and while .NET contains multi-threading, using Unity specific calls outside of the main thread are not permitted. To counter this, I used Loom ("Learn When It's The Right Time", 2011), a plug-in created for Unity that provides a wrapper for C# thread pools and gives the developer a means to indicate, in a given method, which segment of code can run in the main thread and which can run in worker threads.

5.3 Development Activities

As written in subsection 3.5, the development stage spanned activities from February 11th to June 2nd . The following list states the activities that took place in each iteration:

Iteration 1 (11/2 – 17/2) – An initial phase in which the editor's base language and graphical API were chosen according to the comparison of several rapid prototypes made using different candidates.

Iteration 2 (18/2 – 10/3) – Implementation of the GUI's layout including panels, an object drawing system, tool bar and menu buttons as well as viewport and viewport objects' interactions. Additionally, a cursor manager featuring context sensitive cursors, integration of the Petri Net data structure into the editor, the XML layer (reading and writing files along with the conversion of the Petri Nets into PNML) and the undo/redo mechanism were also implemented. Moreover, the validation and simulation mechanism were also introduced in the editor during this iteration as well as some fine tuning in JFern's PNML file creator.

Iteration 3 (11/3 – 5/5) – During this phase the prototype game was designed and implemented alongside the support for input and output places, class nets, transitions' scripts and subnets. Also during this stage, the communication modules for both the editor and the game engine were created.

Iteration 4 (6/5 – 2/6) – Code revisions and interface remodeling were done during this iteration as a result from usability testing.

5.4 Work Management and Prioritization

In order to manage how the development process took place, I devised two lists: a work backlog containing what activities remained and a defect list detailing software defects, found in the application, and their respective severity levels. At the start of each week, I would choose a subset from both lists and divided it into tasks. During the week, I would complete said tasks and, in case all of the them were not completed at the end of the week, the remainder transited over to the next week. Otherwise, I would select another subset from the aforementioned lists. After each activity was implemented, I would perform tests to access if it was defect-free. It is worth mentioning that the work backlog would be revised in order to add or remove elements. Examples of both these lists are stated in appendix A.

Prioritization was given according to the following order: high priority defects, tasks that remained from earlier weeks, medium priority defects, tasks that can be tested independently or tasks that serve as support for other tasks, tasks that are dependent from other tasks and low priority defects. Defect prioritization was attributed in line with their capability to crash or hinder other application functions.

5.5 Implemented Features

As presented through this section, all of the features stated in subsection 4.4 were implemented. However, since there was excess of time during the development stage, two additional features were also implemented:

- Model editing while simulation is running. This feature was thought to increase the application's utility as designers can test whether a given construct works best or not while the game is running. This also promotes a faster game balance process and behavior visual debugging.

- The execution engine can run in batch mode. This is important for scenarios where Petri Nets are not intended to be edited so the application does not need to waste resources in rendering the GUI.

6. Evaluation

6.1 Usability Tests

These evaluations were concentrated on the interface as it was a crucial part of the application and encompassed most of the required attributes and objectives devised for the project. The type of tests chosen to evaluate the interface were formal usability lab tests (Sharp, Rogers, & Preece, 2002) because they provide a means that, besides recording usability errors, result in feedback in terms of user experience and, while though these tests are, by definition, conducted in a controlled environment, the area and conditions in which they took place resemble actual application scenarios.

There were two main goals behind the usability tests. The first one was to identify potential usability problems existent within the interface. The second goal was to assess if whether or not users, experienced or not, could create a video game using the application.

6.1.1 Test Setup

As previously written, the application's interface was evaluated by means of formal usability lab tests. These tests were made in the Information System Research Methods belonging to the Information Systems Group Lab (CISUC) and had an expected time of completion of approximately 1h30m, however subjects were free to surpass this schedule.

These tests consisted in individual sessions where each voluntary tester was prompted to setup and define the behaviors in a video-game using the thesis' application in conjunction with the Unity game engine and pre-existing graphical assets (level and character 3D models). During those sessions, testers were accompanied by an evaluator, I, whose job was to clarify any rising questions and to take notes of events that could happen during the test. In order to help document any event, that ultimately I could not note due to lack of writing speed, audio was recorded. Initially these audio recordings were complemented by a desktop recording but, because of technical problems, most recording files were corrupted and had to be discarded from the results' analysis.

Each session followed a predefined script. Primarily, test subjects were introduced to the project's context and test objectives. Secondly, they were asked demographic questions for later analysis of the population performing the test; the actual test started afterwards, when testers were given a document with information regarding Petri Nets and were encouraged to explore the interface for 5 minutes, after which they were given the game's design document and a list of tasks that contributed for the creation of said game. After each task, test subjects estimated its difficulty in a scale of 1 to 5. Subsequently they were interviewed to detail their overall user experience and performance and were requested to list the top 5 best and worst aspects of the interface, according to their opinion.

The game that test subjects were supposed to create was, as stated, described in a pre-made design document and detailed in a task-list. This document defined the context of the game, its rules, actors and sensors and scripts that were available to them. The task-list helped guide the users in the completion of the game by dividing it into tasks. The first half of the list contained a step-by-step guide while the second half was only comprised of textual descriptions. This way, testers, during the first half, could learn the basics of the application as well as its quirks. An overview of the tasks that subjects had to complete is as follows:

1. Read information regarding Petri Nets and explore the editor for 5 minutes.

2. Read the game's design doc and start a new project.
3. Build a chronometer mechanism.
4. Create an overall score update mechanic when the chronometer reaches zero.
5. Build the player's navigation system.
6. Create the player's shooting mechanism.
7. Make a bot spawning mechanic.
8. Devise the enemies' AI.
9. Integrate the overall score update with a local scoring system.
10. Make a winner announcement system.
11. Save the project.
12. Open a project and answer some questions regarding the language's syntax and semantics.

More detailed information regarding the task-list's tasks can be found in appendix C.

In sum, the game consisted in a competitive first person shooter where players and AI-controlled bots had to toss balls at each other in other to increase their team's score. The game's design doc is presented in appendix B. The following screenshot illustrates the game, as made by one of the test subjects.

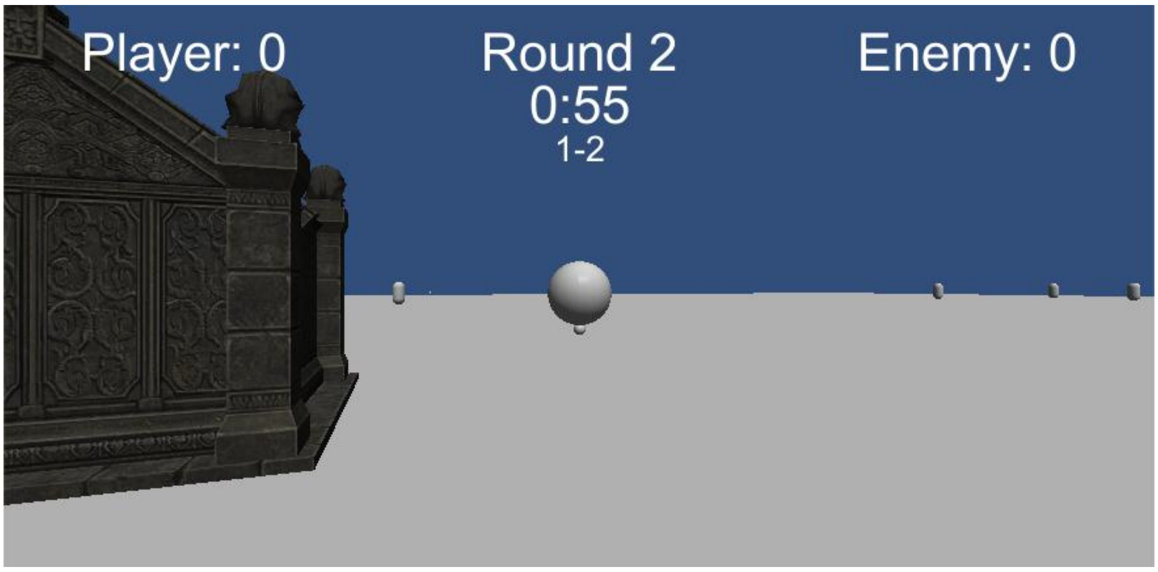


Figure 31 Screenshot of the test game "Spheres of Steel" as created by one of the subjects

Tests were performed using 11 subjects (Sauro, 2011) and the data collected from the recorded audio, demographic questionnaire, interviews and notes, was categorized into three classes: demographic information, user performance and usability issues. Although 11 testers participated, only 10 completed the development of the game and, therefore, the information aggregated from the subject who had to abandon the test midway, due to personal reasons, was only used in the demographic information and usability issues as there was not enough information necessary to compile in the user performance category.

Demographic questionnaires required subjects to state their age, sex and highest academic degree. They also inquired users to rate their experience, in a scale of 0 to 2 – 0 meaning never heard of the term and 2 denoting highly proficient - in textual programming (TP), visual programming (VP) and game development (GD). The reason for this is that textual programming introduces people to algorithms; Experience in visual programming would make the subjects used to the mannerism required to manipulate a visual language; and experience in game development would make users accustomed to the steps involved in creating a game. Table 24 in appendix E depicts the compiled information.

The population sample is composed of 2 females and 9 males, with an average age of 25.63 ± 4.18. Their qualifications range from Bsc student to Phd student. From this information, it can be deducted that the highest degree achieved by the test subjects range from High School to a Msc coinciding with the academic education that game designers often have.

The average experience in textual and visual programming and game development is, as stated, 1.18 ± 0.87, 0.45 ± 0.69 and 0.72 ± 0.66 respectively. This means that subjects are familiar but not proficient in textual programming, barely know about visual programming but have little knowledge of game development.

Overall, the subjects constituting the population sample were selected in a manner that allowed for a heterogeneous sample, in means of qualifications and experience levels. This way, in theory, would increase the amount of issues found by subjects.

6.1.2 Results and Analysis

6.1.2.1 User performance

In this context, user performance consists in the overall time subjects took to complete the game's development and each individual task and their relative perception of the difficulty of every task. This was extrapolated from the audio recordings and ratings that testers gave after finishing tasks. From the matrix in Table 25 two variables were derived: total time per user, calculated by summing the matrix's columns (Figure 32) and the average time per task by averaging its rows (Figure 33). These were used as auxiliary variables to deduce other expressions (see below). Likewise, using the matrix in Table 26, by averaging the columns, the average difficulty per user was calculated and plotted in Figure 34. In the same manner, averaging the rows gives the average difficulty per task as expressed in Figure 35.

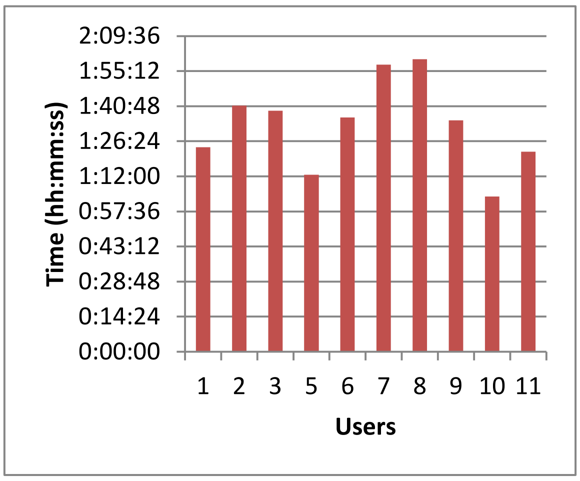


Figure 32 Total Time per User

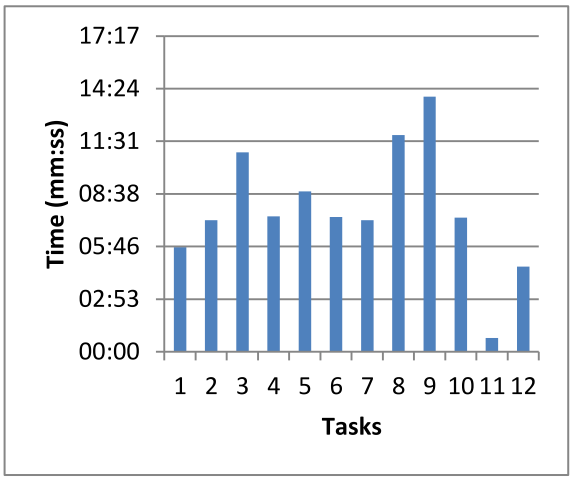


Figure 33 Average Duration per Task

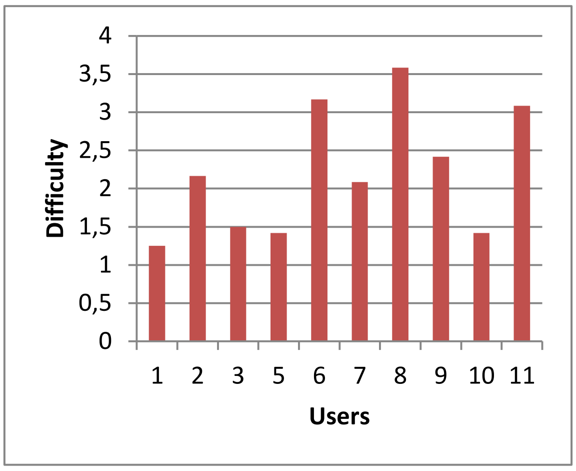


Figure 34 Average Difficulty per User

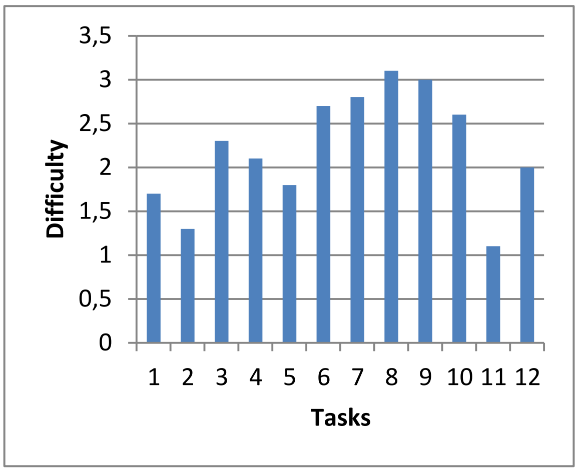


Figure 35 Average Difficulty per Task

From the data presented in Figure 32 it was concluded that on average, testers completed the test in 1h33m09s ± 3m06s, only 3m09s above the expected time and their perception of the test's difficulty was, on average 2.2 ± 0.65 – this was derived from the values available in Figure 34. This means that users thought the test they made, while using the application, was relatively easy. Nevertheless, only 1 out of 10 subjects did not require the evaluator's assistance as evidenced in Table 26.

The average difficulty per task behaves quite as expected. By examining Figure 35 Average Difficulty per Task we can observe that the easiest tasks were 1, 2 and 11. These tasks consisted respectively in exploring the editor, opening a new project and exporting it and, therefore, required minimal effort in comparison to other tasks. It is also noteworthy that task 1 has a higher difficulty value than the remainder 2. This could be explained by the fact that subjects were given a context-free timeslot to explore an application which they had not manipulated before. When testers started to create Petri Nets, i.e. in task 3, difficulty rose as anticipated and, as the users got costumed to the interface, their perceived difficulty decreased progressively during tasks 4 and 5 as the mechanics were similar to those necessary in task 3. During task 6, test subjects did not have a step-by-step guide as in previous tasks but had, in its place, a textual description of the game mechanic they were assumed to implement. As a result, difficulty increased, as anticipated. For tasks 7, 8, 9 and 10 an analogous behavior to tasks 4 and 5 was assumed, yet difficulty rose instead. This could be explained by the fact that these tasks had more complex structures, than task 6, and consequently depended upon additional problem solving expertise. However, tasks 9 and 10 have their average difficulty reduced which could be explained by the fact that testers were now comfortable with the editor's quirks and how to translate the task-lists' tasks into Petri Nets.

6.1.2.2 Usability Issues

The notes and interviews originated in a list of usability issues. These issues, after compiled and normalized, were categorized according to their importance (Sauro, 2011), occurrence frequency, type and occurrence by task and by user. From the usability tests, 406 issues, distributed across 88 different events, were found. A table detailing these issues is presented in appendix D.

There were only three importance levels given to issues: High, Medium and Low. These levels were attributed according to the issue's degree of prevention in completing a certain task. Each level corresponded to a number: High corresponded to 1, Medium to 0.66 and Low to 0.33.

In total, there were 9 types used to classify the issues (Hourcade, 2006). These types were Functional Error (FE), Affordance (A), Feedback (FB), Perception of System State (PSS), Naming Interpretation (NI), Instruction Interpretation (II), Representation Interpretation (RI), Mappings (M) and Domain Knowledge (DK).

Instead of compiling the findings into a table, the following pie-chart showcases the distribution of these types according to the events found.

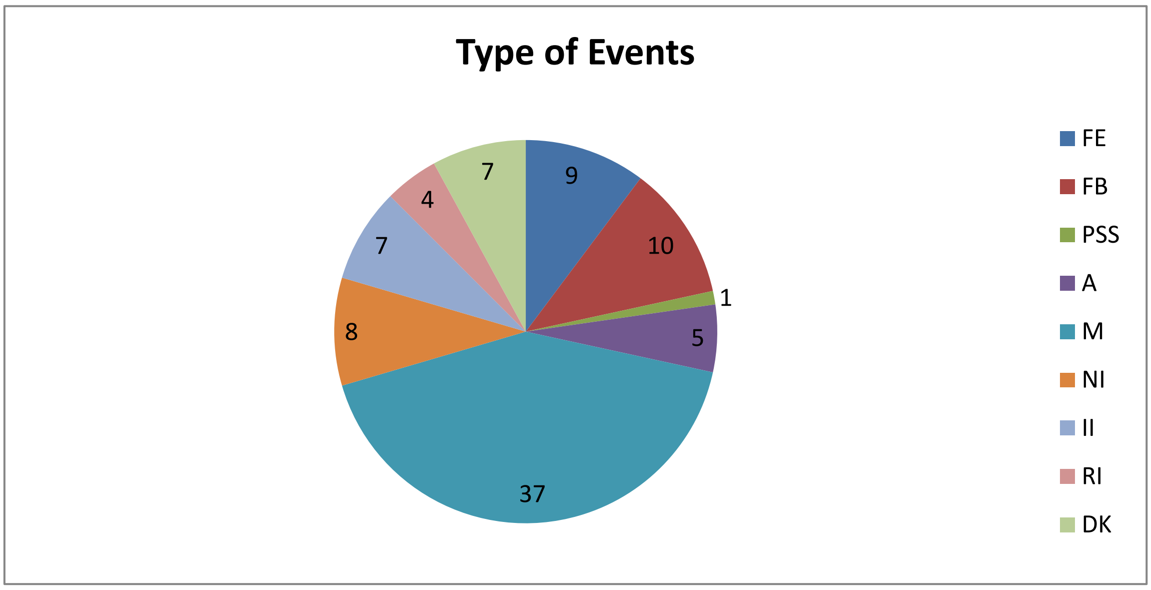


Figure 36 Frequency of issue types

The issue occurrence by task and by user was reported in a task-by-problem and user-by-problem matrices (Sauro, 2012b). From these matrices, it was possible to calculate the average problem frequency by user and task, which was used on a metric explained in subsection 6.1.3, the percentage of problems found by only one user and the percentage of problems found by users and tasks. The matrices are displayed in tables F and G, but due to their size, they had to be split in half.

The matrices in appendixes G and F allowed to assess the problem frequency and percentage of errors found per task and user respectively. These calculations were then plotted for easier visualization. The former are present in figures Figure 37 and Figure 38 and the later in Figure 39 and Figure 40. These values were important during the error correction iteration of the application's editor.

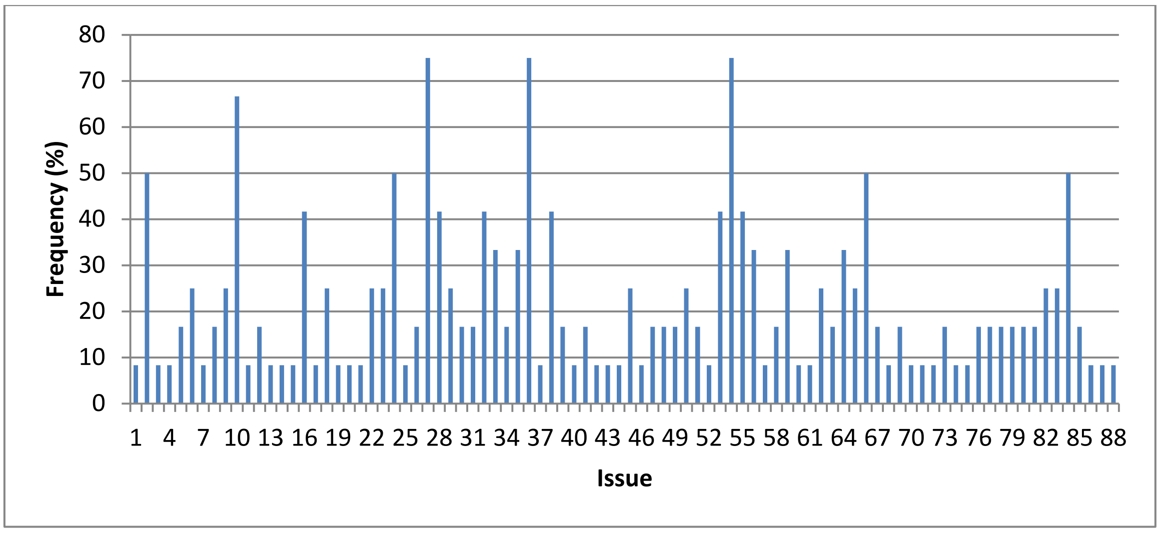


Figure 37 Issue Frequency by Task

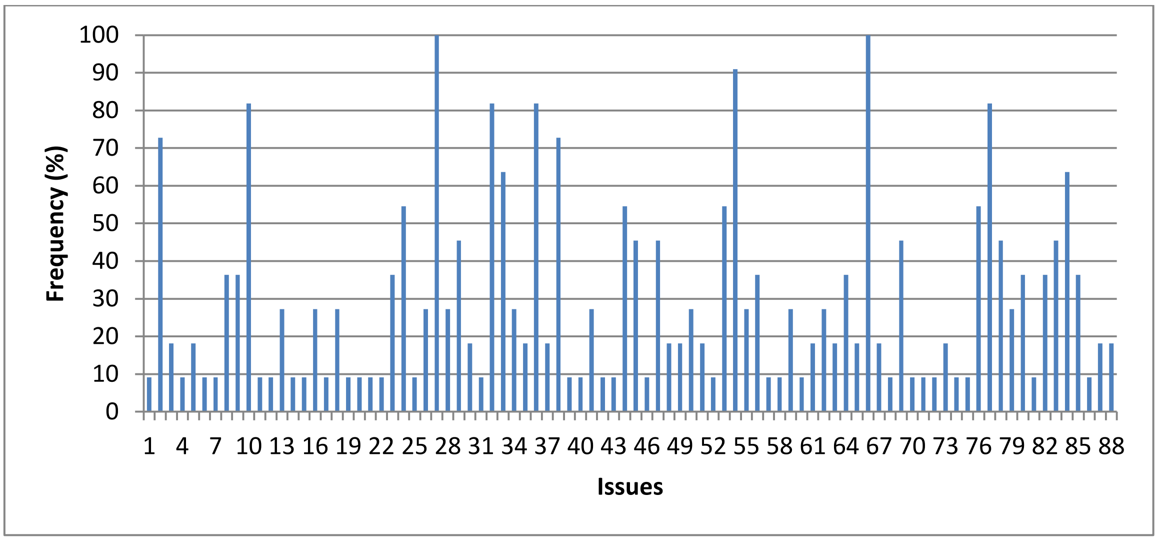


Figure 38 Issue Frequency by User

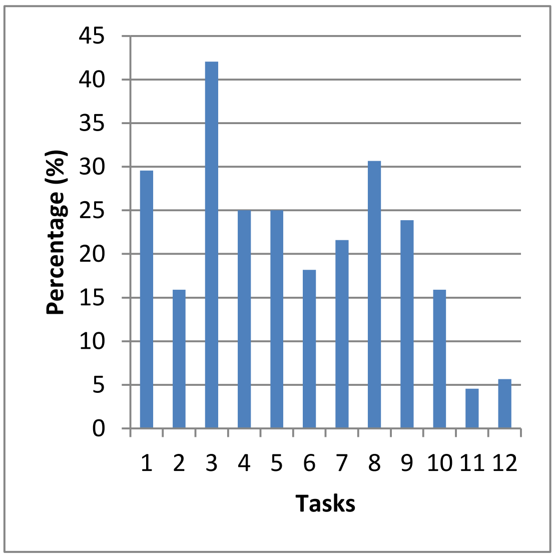


Figure 39 Percentage of errors found per task

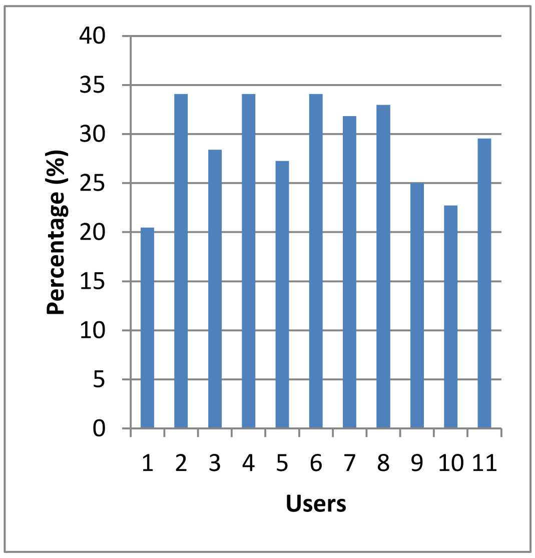


Figure 40 Percentage of errors found per user

Finally, Figure 41 illustrates the relative frequency obtained for the number of occurrences for each issue, which can be consulted in appendix D.

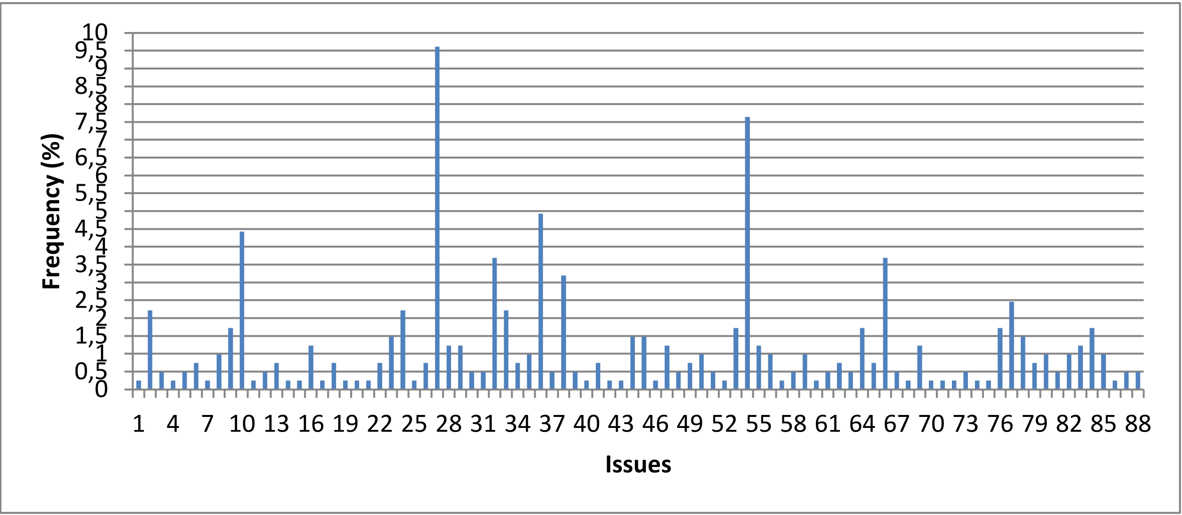


Figure 41 Relative frequency of issue occurrence

From the pie chart in Figure 36, it can be assessed that most events lie on the category of Mappings (Hourcade, 2006). This means there during the tests, most recorded events were comprised discrepancies between their users' intentions and the interface's available actions.

Using this test setup, it can be concluded that most errors were found during the completion of task 3 (approximately 42% of the errors were found there), as shown in Figure 39. This may due to task 3 introducing most of the actions that are used through the rest of the test. From the task-by-problem matrix (appendix G), it can be assessed that 36% of the problems were only encountered once on all 12 tasks (this accounts for about 32 problems). On average, approximately 21% ± 10.5% of the errors were encountered per task. Regarding issue recurrence per task, on Figure 37 it is evident that only 8 out of 88 (roughly 9%) of the issues appear at least in half of the tasks.

In Figure 40 it is revealed that each user found nearly the same amount of problems. In fact, on average, they encountered 29% ± 4.6% of the total errors. As similar to problems that appeared only on one task, 36% of the issues were also found by a single user. However, unlike the most frequent issues per task, there were 15 errors that were found by more than half of the task subjects, as is depicted in Figure 38.

6.1.3 Error Corrections

As stated, one of the usability tests' goals was to discover usability errors present in the editor so that it could be corrected. Yet, due to time constraints and for the fact that some issues were mutually exclusive, meaning that fixing one problem could perpetuate another, not all issues could be addressed. Instead, by applying the Pareto Principle (Sauro, 2012a) to the list of errors, in theory, correcting the equivalent to "20%" would account for a corresponding "80%" of all bad interactions. With that in mind, I devised a metric to help identify the most critical problems and to sort accordingly so that I could elaborate a correction plan and consequently fix most, if not all, of the critical errors.

This metric consisted in a two part algorithm. In the first part, a value, referred to as priority level from now on, was attributed to each issue by calculating the arithmetic product between the its frequency per user (IFU) and per task (IFT), its relative frequency (RF) and its importance (I). The formula is given by the expression $Priority_i = IFT_i \times IFU_i \times RF_i \times I_i$. This results in a value ranging from 0 to 1 because all of the above-mentioned variables were normalized beforehand. By multiplying these factors, it is assured that, for instance, issues that appeared frequently during tasks, were encountered by most users and tasks, and hindered the completion of said tasks are given more priority than issues that, for example, were not as frequent or important. A chart detailing the priority levels per problem, sorted by value, is presented in Figure 42 Priority Level per Issue.

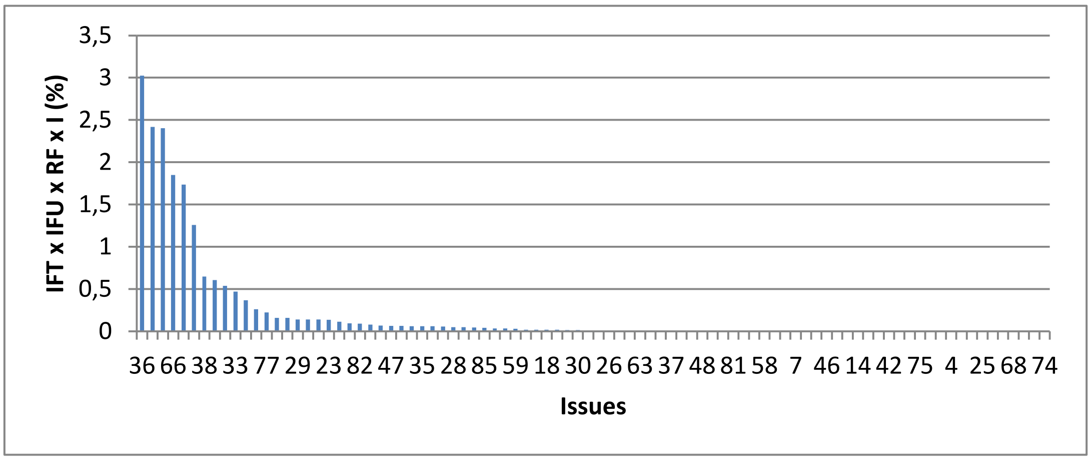


Figure 42 Priority Level per Issue

The second step of the algorithm required introducing the resulting values in a matrix and color scaled its cells. This means that cells were given a color in a white-red gradient according to their distance to the minimum and maximum calculated values (0.0006 and 3.0228 respectively). Cells closer to the minimum were painted white whilst the others were progressively colored red. Appendix H presents the calculated priority levels in a color scaled matrix. Much like the problem-by-user and problem-by-task matrices, this one had to be split in order to fit on a page as well.

From the table, I selected the issues whose cell color included shades of red and sorted them by their hue (bright red as top priority and light pink as low priority). These cells corresponded to the left most issues illustrated in Figure 42.

In appendix D, problems that contain a green cell means that they were considered to be resolved. However, before resolving said issues, events of type DK and II were discarded because although they were documented during the tests, they were not related to the application per se, but to the subject's inherent domain knowledge and interpretation skills. Thus, the only solution to correct these issues would be to provide better information and a task-list written more clearly. Another issue that was discarded was issue 85. The reason behind this was that the introduction of a new structure, an "if" transition, would require a revision of the editor's language's grammar in order to introduce flow control mechanisms which was already considered for future work (see subsection 7.2). How the issues were solved is also present in appendix D.

The revised interface is illustrated in the screenshot in Figure 43.

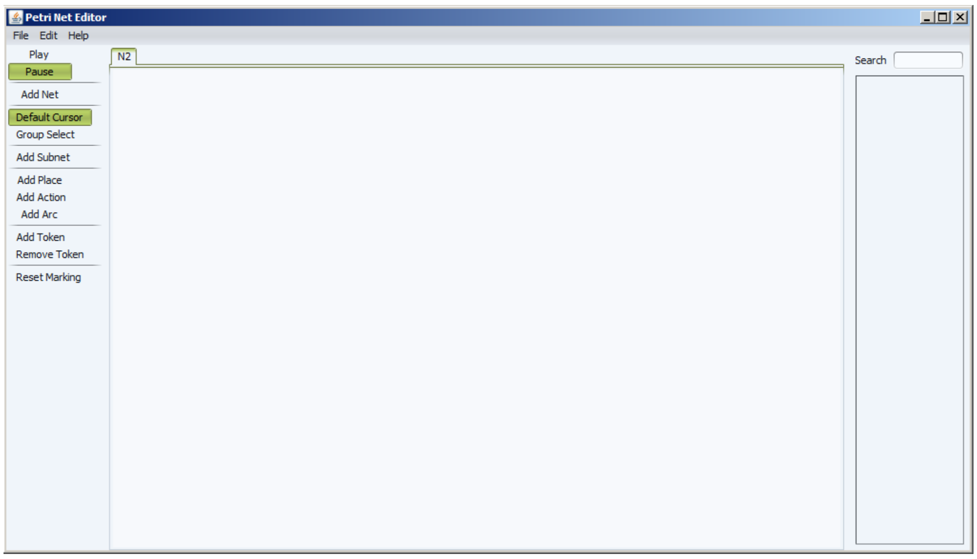


Figure 43 Screenshot of the editor interface (post error correction)

6.2 Performance Tests

Since scalability was one of the attributes required for the application, some performance tests were made in order to verify that the editor and execution mechanism could endure a large number of actors performing simultaneously on a stage. This gave an idea on how the application could behave on single player games.

Unfortunately, testing the scalability regarding the number of connections, i.e. players, was not feasible as MMO scenarios contain a large number of simultaneous players which would require resources that were unavailable. Furthermore, time constraints did not allow this type of tests to be conducted.

6.2.1 Test Setup

Unlike usability tests, these were not executed with test subjects. Instead, they consisted in running the editor alongside a modified version of the proof of concept game Orphibs. This mod added a button to spawn actors on demand without additional machines connecting to the game.

While the game was running, the spawn button was pressed in order to create as many actors as possible until the game was left unplayable due to increased lag. During this time, some attributes of both applications were measured using a resource monitor. These attributes included the applications' average CPU and GPU, physical memory and total network usage. Additionally, the number of spawned actors was also documented.

For further comparison, these tests were executed 3 times using two different configurations: one with the editor in batch mode and the other with it in graphic mode. This allowed to evaluate if the editor's renderer would impact its performance.

The machines' hardware utilized for these tests was: a desktop computer with an Intel Core 2 Quad @2.33GHz processor, 4Gb of RAM, running a 64-bit version of Windows 7 Ultimate, NVidia GeForce 9600 GT graphics card and a connection speed of 100Mb/s and a laptop with an Intel Core i7 CPU Q720 @1.60GHz processor, 6Gb of Ram, running a 64-bit version Windows 7 Ultimate and a NVidia GeForce GT 330M graphics and a connection speed of 54Mb/s. It is worth mentioning that the former machine was used to execute Unity and the editor was deployed on the later.

6.2.2 Results and Analysis

Table 23 illustrates the results achieved during the scenarios:

Table 23 Average Resource Usage per Test Scenario

Tests		Unity				Editor			
Configurations	Number of Actors	CPU (%)	Physical Memory (KB)	Total Network (B/s)	GPU (%)	CPU (%)	Physical Memory (KB)	Total Network (B/s)	GPU (%)
	2	23	88	156	8	4	123	176	0
Batch Mode	72	29	109	264	3	3	93	258	0
	122	27	119	256	5	3	94	253	0
	2	24	87	166	8	14	209	161	24
Graphic Mode	72	28	105	264	4	13	127	263	19
	122	27	128	261	4	12	128	259	21

As can be observed, in both configurations, Unity's CPU usage grows slightly when increasing the number of actors but then stabilizes around 27-29% unlike the editor which has a stable usage of 3-4% and 12-14% on each scenario respectively (however, 12% is high for that particular processor as each core on the editor's machine corresponds to 12.5% of the total usage). This means that, even though the editor running in graphics mode occupies one of the CPU's core, increasing the number of actors has a bigger impact on Unity. One thing worth noting is that while Unity stops working at around 122 actors, its CPU usage is only 27%. This is because Unity's default frame rate configuration allows the game to render as fast as it can (Unity Technologies). Since the engine is single threaded (with additional threads created by me for the communications module), the main thread occupies one of the processor's four cores entirely as each core is related to 25% of total usage.

Regarding physical memory, Unity's usage increases at approximately the same rate in both scenarios. The editor, on the other hand, decreases its memory. An explanation for this event is that most of the objects created initially are purged by the garbage collector later on. Nevertheless, as expected, running the editor in graphics mode has more memory usage than running it in batch mode due to the fact that the visual editor requires to maintain GUI structures in memory.

From all of the attributes, GPU has a more stable behavior in both applications and scenarios. However, the editor has surprisingly more GPU usage running a 2D application than Unity has running a 3D game.

Finally, network usage increases roughly in the same manner in both applications and scenarios. There is a growth in network traffic when the applications have to account for 72 actors but it then stabilizes subsequently.

In conclusion, it is apparent that Unity is a bottleneck as it did not allow to add more actors to the stage whilst the editor maintained functions. However, this is not critical for most genres because the number of simultaneous actors present in a given stage is usually below

122 but for some genres including multi-agent simulations, MMO or even Real Time Strategy, in which the number of simultaneous actors can reach thousands, having a maximum number of 122 actors is not ideal. Another issue discovered from these tests is that the editor's GUI module needs to be optimized in order to reduce GPU, and possibly CPU, usage. In these conditions, editing a big Petri Net could become unfeasible.

7. Further Work

7.1 Critical Aspects to Correct

There were some aspects regarding the application that were left to be corrected as future work: Firstly, due to lack of further knowledge at the time of implementation, the thread safe net manipulator uses the "synchronized" keyword for resource locking in order to avoid unsafe operations. Using this keyword has some disadvantages as this method does not allow some resource locking functionalities such as try lock. With the purpose of circumventing this limitation, some of the coding was done in a way that could jeopardize thread safe operations (although this was not confirmed during any of the tests). It would be best to rewrite the thread safe net manipulator module using Java's Lock object that was introduced in this language's most recent versions as it ensures more features than the "synchronized" keyword. Secondly, it was found, at the time of writing, that places having neutral tokens that are meant to be shared across multiple actor instances will cause race conditions. This means that the first actor to enable a transition, in which a place with a neutral token is linked, will cause other actors to be unable to fire said transition. Introducing the concept of shared places where each neutral token is multiplexed according to the number of actors would solve this issue. Another aspect was found during performance tests. By observing their results, it was concluded that although the editor was not a bottleneck, its graphics mode was not optimized regarding CPU and GPU usage. Reducing draw calls could diminish the amount of usage and thus resulting in an editor more CPU and GPU friendly. It was also assessed from these tests that Unity was CPU intensive because it was configured to render as fast as it could. By imposing a fixed frame rate, Unity would then decrease its CPU usage which in turn could potentially reduce, or even remove, the bottleneck found. Finally, visual aesthetics were not considered during the development of the prototype. It would be interesting to make the editor have a look and feel more oriented towards its target audience.

7.2 Future Developments

One of the project's quality attribute was scalability regarding the number of players which, due to lack of resources and time could not be evaluated. For future developments, it would be interesting to validate this attribute, through benchmarking, as it would mean that the application could endure MMO scenarios. Another interesting activity would be to refine the interface further using additional usability tests. This way, corrections made previously would be validated and other aspects of the interface could be tested in a more in-depth manner.

After that, two possibilities were discussed: iterate the language by adding new elements, for instance flow control transitions, colored tokens (Aalst, 2011), and turn the existing transitions into functions instead of script executors. The overall result would, theoretically, lead the language to become closer to a programming language and increase its expressiveness. The other possibility was to append a content generating mechanism to the editor. This generator would create Petri-Nets and, through an activity logger feeding player activity, would readjust existing nets resulting in a procedural game-design done by feedback-reinforcement.

8. Conclusions

In this thesis, I presented a solution proposal to counter the downsides of game programming and to allow designers to build behavior and choreography models for actors in game/simulation contexts. This proposal consisted in a language specification, based on Petri Nets, a system encompassing a visual editor, an interpretation/execution engine and a communication mechanism that allowed the translation between modeled behaviors and in-game actions on arbitrary game/simulation engines. This system was also devised to support MMO due to its popularity as a game mode.

To validate this proposal, a proof of concept game prototype was created and it was proven a success as the game's actors, entirely modeled using the application functioned as intended whether in single, or multiple, machines. Additional validation was made using usability tests which were also proven to be successful as, on average, the heterogeneous population sample (with different levels of experience regarding textual programming, visual programming and game development) found the editor easy to use. Furthermore, some scalability tests regarding the number of simultaneous actors were conducted. It was then concluded that although the game engine presented a bottleneck that limited the number of actors present, overall results indicate that the current configurations allow to develop most game genres. However, due to time and resource constraints, additional scalability tests, regarding the number of concurrent connections/players could not be performed. In sum, the application was proven to suffice the devised objectives and, as scientific contribution, the interaction design model, architectural proposal and application were originated from this research.

During the course of this thesis' research and development I had the opportunity to deepen my knowledge base concerning several computer engineering topics. Most notably, I have gained more information regarding Petri Nets: how do they work and what are their state of the art applications in game development; a new research method that promotes agile development; I have learned how MMO architectures are designed; The multi-thread and network programming I had to do allowed to consolidate and extend my shallow knowledge in those areas; Finally, I discovered how to devise and conduct usability tests which turned out to be more difficult to perform than I had anticipated. Nevertheless, the practice I gained will allow me to execute better tests in my professional future. In conclusion, this thesis has allowed me to grow my skills in order to become a better professional.

References

Learn when it's the right time to use threads for your game and get to grips with some of the complexities they impose. See how easy it can be to add a multithreaded functions to your game... (2011). *Unity Gems*. Retrieved March 23, 2013, from http://unitygems.com/threads/

AALborg University in Denmark. (2012). *TAAPAL (Version 2.2.1) [Computer application software]*. Retrieved February 25, 2013, from http://www.tapaal.net/introduction/

Aalst, W. (2011). Hierarchical Petri-Nets. Eindhoven University of Technology. Retrieved May 4, 2013, from http://cpntools.org/_media/book/hcpn.pdf?

Adobe Systems Software Ireland Ltd. (2012). *Flash Professional CS (Version 6) [Computer application software]*. Retrieved October 3, 2012, from http://www.adobe.com/products/flash.html

AgentSheets Inc. (2010). *AgentSheets (Version 3.0) [Computer application software]*. Retrieved October 3, 2012, from http://www.agentsheets.com/

Araújo, M., & Licinio, R. (2009). Modeling Games with Petri Nets. *Digital Games Research Association (DiGRA) on Innovation in Games, Play, Practice and Theory.*

Axelrod, R., & Amir, G. (2012). *Massively Multiplayer Game Development 2: Architecture and Techniques for an MMORTS*. Retrieved October 29, 2012, from Gamasutra: http://www.gamasutra.com/view/feature/130738/massively_multiplayer_game_.php?print=1

Bandini, S., Manzoni, S., & Vizzari, G. (2004). Situated Cellular Agents for Crowd Simulation and Visualization. *iMES conference of Complexity and Integrated Resources Management.*

Baptista, T., & Costa, E. (2008). Evolution of a multi-agent system in a cyclical environment. In *Theory in Bioscience* (pp. 141-148). Berlin: Springer-Verlag. doi:10.1007/s12064-008-0031-2

Barbosa, A., & Azevedo, M. (2013). *JPetriNet (Version 1.1) [Computer application software]*. Retrieved February 25, 2013, from http://jpetrinet.sourceforge.net/

Billington, J., Christensen, S., van Hee, K., Kindler, E., Kummer, O., Petrucci, L., . . . Weber, M. (2003). The Petri Net Markup Language: Concepts Technology and Tools. In *Applications and Theory of Petri Nets 2003* (pp. 483-505). Berlin: Springer-Verlag.

Bonet, P., Lladó, C., & Puigjaner, R. (2007). PIPE v2.5: A Petri Net Tool for Performance Modelling. *23rd Latin America Conference on Informatics (CLEI 2007).*

Brom, C., & Abonyi, A. (2006). Petri Nets for Game Plot. *AISB on Narrative AI and Games workshop.*

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (2011). Reflection. In *Pattern-Oriented Software Architecture: A System of Patterns* (pp. 193-219). Croydon: Wiley.

Catto, E. (2011). *Box2D (Version 2.2.1) [Computer application software]*. Retrieved October 3, 2012, from http://box2d.org/

Cooperative State University Karlsruhe. (2012). *WoPeD (Version 3.0.1) [Computer application software]*. Retrieved February 18, 2013, from http://woped.ba-karlsruhe.de/

Crytek. (2011). *CryEngine (Version 3) [Computer application software]*. Retrieved October 3, 2012

Dingle, N., Knottenbelt , W., & Suto, T. (2009). PIPE2: A Tool for the Performance Evaluation of Generalised Stochastic Petri Nets. *ACM SIGMETRICS Performance Evaluation Review, 36*(4), pp. 34-39.

Dormans, J. (2009). *Machinations Framework*. Retrieved January 1, 2013, from Machinations Wiki: http://www.jorisdormans.nl/machinations/wiki/index.php?title=Machinations_Framework

Douceur, J., Lorch, J., Uyeda, F., & Wood, R. (2007). Enhancing Game-Server AI with Distributed Client Computation. *17th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*.

Electro Tank. (2012). *ElectroServer 5*. Retrieved October 29, 2012, from http://www.electrotank.com/es5.html

Enterbrain Inc. (2005). *RPG Maker (Version XP) [Computer application software]*. Retrieved October 3, 2012, from http://www.rpgmakerweb.com/

Epic Games. (2012, July). *UDK (Version 3) [Computer application software]*. Retrieved October 3, 2012, from http://www.unrealengine.com/udk/

Exit Games. (2012). *Photon Server*. Retrieved October 29, 2012, from http://www.exitgames.com/Photon/

Fischer, E. (2002). *PetriNet Kernel (Version 2.2) [Computer application software]*. Retrieved February 25, 2013, from http://www2.informatik.hu-berlin.de/top/pnk/

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (2011). *Design Patterns: Elements of Reusable Object-Oriented Software*. Westford: Addison Weasley.

gotoAndPlay(). (2012). *SmartFoxServer*. Retrieved October 29, 2012, from http://www.smartfoxserver.com/

Hampel, T., Bopp, T., & Hinn, R. (2006). A Peer-To-Peer Architecture for Massive Multiplayer Online Games. *5th ACM SIGCOMM workshop on Network and system support for games*.

Havok.com Inc. (2012). *Havok Tool Suite*. Retrieved October 3, 2012, from http://www.havok.com/products

Heiner, M., Herajy, M., Liu, F., Rohr, C., & Schwarick, M. (2012). Snoopy – a unifying Petri net tool. In *Applications and Theory of Petri Nets 2012* (pp. 398-407). Berlin: Springer-Verlag.

Hevner, A., & Chatterjee, S. (2010). The General Design Cycle. In *Design Research in Information Systems: Theory and Practice* (pp. 26-27). Berlin: Springer-Verlag.

Hourcade, J. (2006). Usability Principles. University of Iowa. Retrieved May 16, 2013, from https://www.cs.uiowa.edu/~hourcade/classes/fa06hci/lecture2.pdf?

Imperial College London. (2012). *PIPE2 (Version 4.2.0) [Computer application software].* Retrieved November 5, 2012, from http://pipe2.sourceforge.net/

Jenkins Software LLC. (2012). *Raknet.* Retrieved October 29, 2012, from http://www.jenkinssoftware.com/

Kadlec, R. (2008). Evolution of intelligent agent behavior in computer games. Charles University in Prague. Retrieved January 17, 2013, from http://artemis.ms.mff.cuni.cz/main/papers/

Kahn, K. (1995). Metaphor Design: Case Study of an Animated Programming Environment. *Computer Game Developer Conference.*

Kahn, K. (1996, August). Drawings on Napkins, Video Game Animation, and other ways to Program Computers. *Communications of the ACM, 39*(8), pp. 49-59. doi:10.1145/232014.232028

Kahn, K. (1996). Seeing Systolic Computations in a Video Game World. *IEEE Conference on Visual Languages.* doi:10.1109/VL.1996.545274

Kahn, K. (2000, March). Generalizing by Removing Detail: How Any Program Can Be Created by Working with Examples. *Communications of the ACM, 43*(3), pp. 104-106.

Kahn, K. (2006). Time Travelling Animated Program Executions. *Software Visualization Conference.*

Kahn, K. (2009). *ToonTalk (Version 3) [Computer application software].* Retrieved October 3, 2012, from http://www.toontalk.com/

LAAS/CNRS. (2013). *Tina (Version 3.1.0) [Computer application software].* Retrieved February 18, 2013, from http://projects.laas.fr/tina/download.php

Lantinga, S. (2012). *SDL (Version 1.2.15) [Computer application software].* Retrieved October 3, 2012, from http://www.libsdl.org/

Lim, M., Dias, J., Aylett, R., & Paiva, A. (2009). Intelligent NPCs for Educational Role Playing Game. In *Agents for Games and Simulations* (pp. 107-118). Berlin: Springer-Verlag.

Lohmann, N., Mennicke, S., & Sura, C. (2010). The Petri Net API: A collection of Petri net-related functions. *17th German Workshop on Algorithms and Tools for Petri Nets (AWPN 2010).*

Mateas, M., & Stern, A. (2005). Procedural authorship: A case-study of the interactive drama Façade. *Digital Arts and Culture (DAC).*

Mateas, M., & Stern, A. (2005). Structuring content in the Façade interactive drama architecture. *First Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE).*

Microsoft. (2010). *Direct X SDK (Version 9.29.1962) [Computer application software].* Retrieved October 3, 2012, from http://msdn.microsoft.com/en-us/library/ee663274%28v=vs.85%29.aspx

Microsoft. (2010). *XNA Game Studio (Version 4.0) [Computer application software].* Retrieved October 3, 2012, from http://msdn.microsoft.com/en-us/library/bb200104.aspx

Microsoft. (2012). *Kodu (Version 1.2.38.0) [Computer application software]*. Retrieved October 3, 2012, from http://research.microsoft.com/en-us/projects/kodu/

Milligton, I., & Funge, J. (2009). *Artificial Intelligence for games* (2nd ed.). Burlington: Morgan Kaufmann.

MIT Media Lab. (2009). *Scratch (Version 1.4) [Computer application software]*. Retrieved October 3, 2012, from from http://scratch.mit.edu/

Müller, J., & Gorlatch, S. (2004). A Scalable Architecture for Multiplayer Computer Game. *Informatik 2004 conference.*

Nadle, D. (2003, April). A C++ Petri Net Framework For Multithreaded Programming. *ACCU Overload Journal*(53). Retrieved November 5, 2012, from http://accu.org/index.php/journals/357

Natkin, S., Vega, L., & Grünvogal, S. (2004). A new methodology for Spatiotemporal Game Design. *CGAIDE'2004, Fifth Game-On International Conference on Computer Games: Artificial Intelligence.*

Nowostawski, M. (2013, February 22). *JFern (Version 4.0) [Computer application software]*. Retrieved from http://sourceforge.net/projects/jfern/

Petri, C., & Reisig, W. (2008). *Petri Net.* Retrieved September 9, 2012, from Scholarpedia: http://www.scholarpedia.org/article/Petri_net

Procedural Arts. (2005). *Façade (Version 1.1) [Video-Game]*. Retrieved January 13, 2013, from http://www.interactivestory.net/

Rausch, M. (1998). AgentSheets – Programming above C-Level. *Computer Graphik Topics, 10*(3), pp. 10-12.

Repenning, A., & Citrin, W. (1993). AgentSheets: Applying Grid-Based Spatial Reasoning to Human-Computer Interaction. *IEEE Workshop on Visual Languages.* doi:10.1109/VL.1993.269581

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., . . . Kafai, Y. (2009, November). Scratch: Programming for All. *Communications of the ACM, 52*(11), pp. 60-67. doi:10.1145/1592761.1592779

Riesz, M., Baláž, M., & Juhás, G. (2010). PetriFlow: A Petri Net Based Framework for Modelling and Control of Workflow Processes. *Workshops of the 31st International Conference on Application and Theory of Petri Nets and Other Models of Concurrency (PETRI NETS 2010).*

Saltsman, A. (2011). *Flixel (Version 2.55) [Computer application software]*. Retrieved October 3, 2012, from http://flixel.org/

Sauro, J. (2011, July). *10 Things to Know about Usability Problems.* Retrieved May 25, 2013, from http://www.measuringusability.com/

Sauro, J. (2012, September). *Applying the Pareto Principle to the User Experience.* Retrieved May 25, 2012, from Measuring Usability: http://www.measuringusability.com/

Sauro, J. (2012, June). *Report Usability Issues in a User by Problem Matrix.* Retrieved May 25, 2013, from Measuring Usability: http://www.measuringusability.com/

Scalify Pty Ltd. (2012). *Badumna.* Retrieved October 29, 2012, from http://www.scalify.com/

Sega. (2001). *Sonic Adventure 2 [Video-Game].* Retrieved May 5, 2013, from http://www.sega.com/games/sonic-adventure-2/

Sharp, H., Rogers, Y., & Preece, J. (2002). *Interaction Design: beyond human-computer interaction.* United States of America: Wiley.

Shilov, Y. (2013). *Petri Net Editor [Computer application software].* Retrieved February 18, 2013, from http://sourceforge.net/projects/petrineteditor/

Silicon Graphics International Corp. (2012). *Open GL (Version 4.3) [Computer application software].* Retrieved October 3, 2012, from http://www.opengl.org/

Sony Computer Entertainment Europe. (2011). *Little Big Planet 2 [Video-Game].* Retrieved October 3, 2012, from http://www.littlebigplanet.com/2/

Square Enix. (1987). *Final Fantasy [Video-Game].* Retrieved October 3, 2012, from http://na.square-enix.com/games

Stencyl, LLC. (2012). *Stencyl (Version 2.1.0) [Computer application software].* Retrieved October 3, 2012, from http://www.stencyl.com/

Torus Knot Software Ltd. (2012). *Ogre (Version 1.8.1) [Computer application software].* Retrieved October 3, 2012, from http://www.ogre3d.org/

TU Eindhoven & Deloitte. (2005). *Yasper [Computer application software].* Retrieved February 18, 2013, from http://www.yasper.org/

Unity Technologies. (2012). *Unity 3D (Version 4) [Computer application software].* Retrieved October 3, 2012, from http://unity3d.com/

Unity Technologies. (n.d.). targetFrameRate. *Unity Script Reference.* Retrieved June 28, 2013, from http://docs.unity3d.com/Documentation/ScriptReference/Application-targetFrameRate.html

Wang, J. (2002). Petri Nets for Dynamic Event-Driven System Modeling. In *Handbook of Dynamic System Modeling* (pp. 24;1-24;16). Chapman & Hall/CRC.

Ward, J. (2008, April). *What is a Game Engine?* Retrieved October 3, 2012, from Game Career Guide: http://www.gamecareerguide.com/features/529/what_is_a_game_.php

Werf, J., & Post, R. (2004). EPNML 1.1 – an XML format for Petri nets. Retrieved from www.win.tue.nl/~jmw/_media/public/pnmldef.pdf

Woolridge, M. (2011). Practical Reasoning Agents. In *In An Introduction to MultiAgent Systems* (2nd ed., pp. 65-70). Glasgow: John Wiley and Sons Ltd.

World Wide Web Consortium. (2011). *HTML (Version 5) [Computer application software].* Retrieved October 3, 2012, from http://www.w3.org/TR/2011/WD-html5-20110525/

Appendixes

A. Work Management Examples

Work Backlog

- Add support for several independent Petri nets
- Import xml models as subnets
    - import models
    - change ids
- Add sensor channels to XML
- Script integration
    - Add actions to transitions
    - Add input places
    - Add output places
    - Store action script and actor receiver in transitions
- Add curved arrows

Defect List

- Message XSD has syntax errors (High)
- Deleted fusion places are still present in the Petri Net model (Medium)
- Anchor points do not have the right coordinates (Low)
- Mouse events are not firing fast enough (Medium)
- Messages are not being broadcast when necessary (High)
- Dragging an arc issues a weird behavior (Low)
- Cursors' images do not correspond to their functionalities (Medium)

B. Design Document

Introduction

"Spheres of Steel" is a single-player competitive FPS in which the player must toss balls at his opponents and vice-versa. In this game, the player competes against several AI-controlled bots, through several rounds.

Game structure

Control

The game takes place in an open arena in which the player can move freely. The arena is populated by several bots and the player can interact with these bots by tossing balls, from their endless supplies. Bots can also toss balls at the player. If a ball hits an entity, that entity disappears unless the victim is the player. In that case, he will respawn moments later.

Both the bots and the player form each a separate team.
The game is organized in 7 timed rounds that last about 60 seconds each.

Scoring

The game keeps a scoreboard for both teams. Whenever a ball toss is successful (i.e. hits an opponent), score is incremented for the attacking party. At the end of each round, the team with the highest score wins the round. After 7 rounds, the team with most rounds won, wins the game. It is worth noting that the game plays a sound to signal the end of the final round.

HUD

The HUD merely consists of text stating the remaining time as well as the scoreboard. The scoreboard is composed of each team's round and final scores and, additionally, the current round. These texts should be located at the top corner of the screen.

Actions

Movement

The player can move by pressing the arrow keys or "wasd" keys. However, both the left and right keys as well as the "a" and "d" keys will make the player strafe. Rotation is done by moving the mouse sideways.

Fighting

Tossing balls is done by pressing the left mouse button.

AI

There is only one type of bots. When they see the player they follow him. Otherwise, they scan their surroundings. During both phases, they can shoot balls.

Assets

In order to create this game, you must build the logic underlying the behavior of three actor types: Player, HUD and Enemy.

Function Set

There are several available functions that you can use in order to develop this game. These functions are spread across the three actor types.

Player

Move – Moves the player according to keyboard input.

Rotate – Rotates the player according to mouse input.

UpdateScore – Updates the enemies' team score.

Die – Makes the player die.

Respawn – Makes the player spawn in the middle of the level.

Shoot – Makes the player shoot a ball.

Enemy

ScanSurroundings – Makes the enemy rotate on itself.

Seek – Makes the enemy seek the player.

UpdateScore – Updates the player's team score.

Die – Makes the enemy die.

Respawn – Makes the enemy spawn in the middle of the level.

Shoot – Makes the enemy shoot a ball.

HUD

UpdateTime – Updates the HUD's clock.

ResetTime – Resets the HUD's clock.

PlayHornSound – Plays a Horn sound.

AnnounceWinner – Updates the HUD's round text to announce a winner.

CheckRoundWinner – Updates the teams' scoreboard with the round winner.

SpawnEnemies – Eliminates the existing enemies and spawns new ones.

Sensors

In addition, these actor types also have sensors.

Player

Enemyhit – A collision sensor. It is activated when an enemy ball hits the player.

inputaxisvertical – Input sensor. Corresponds to the up/down keys and equivalents.

inputaxishorizontal – Input sensor. Corresponds to the left/right keys and equivalents.

inputmousex – Input sensor. Mouse horizontal movements.

inputmousey – Input sensor. Mouse vertical movements.

inputmousebutton0stay – Input sensor. Activated when the left mouse button is kept pressed.

inputmousebutton1stay – Input sensor. Activated when the middle mouse button is kept pressed.

inputmousebutton2stay – Input sensor. Activated when the right mouse button is kept pressed.

inputmousebutton0down – Input sensor. Activated when the left mouse button is pressed.

inputmousebutton1down – Input sensor. Activated when the middle mouse button is pressed.

inputmousebutton2down – Input sensor. Activated when the right mouse button is pressed.

inputmousebutton0up – Input sensor. Activated when the left mouse button is released.

inputmousebutton1up – Input sensor. Activated when the middle mouse button is released.

inputmousebutton2up – Input sensor. Activated when the right mouse button is released.

Enemy

Playerhit – A collision sensor. It is activated when a player ball hits an enemy.

sees – Enemys' vision sensor, activates when it sees the player.

not sees – Another enemys' vision sensor, activates when it doesn't see the player.

can shoot – This sensor is related to the weapon cool down. When activated, it means the enemy can shoot.

HUD

time – Time elapsed. This sensor is updated every second.

C. Task-List

Task #1: Learn the basics

Your first task is to learn the basics of the editor by exploring it for five minutes.

A. Read the document handed to you.
B. Open the editor.
C. Explore the editor.

Be sure to say what you are looking at and what you are thinking, from now on.

Task #2: Starting out

A. Read the design doc handed to you.
B. Create a blank project.
C. Add 2 more class nets.
D. Name the class nets: “Player”, “HUD” and “Coward”.

Task #3: HUD Clock

A. Switch the current class net to “HUD”.
B. Add an input place for the sensor “time” and name it “Time”.
C. Add a transition called “Update Time” and make it run the script “UpdateTime” and associate it to the “hud” channel.
D. Link, through an arc, the recently added input place and transition.
E. Add another transition named “Reset Time”, making it run the script “ResetTime” on the “hud” channel and create two regular places: “Tick” and “Reset”.
F. Add arcs linking “Update Time” to “Tick”, “Tick” to “Reset Time” and “Reset Time” to “Reset”.
G. Increase the weight of the arc linking “Tick” to “Reset Time” to 60.
H. Start the system (game + simulation) and check if the clock on the screen works.

Task #4: Score board update

A. Switch the current class net to “HUD”.
B. Create a transition “Check Round Winner” with the script “CheckRoundWinner” and “hud” attached.
C. Link “Reset” to “Check Round Winner”.
D. Add a regular place called “ScoreBoard Updated”.
E. Add a transition “Reset Round Score” that executes “ResetRoundScore” on the “hud” channel and a place called “Scores Reset”.
F. Link “Check Round Winner” to “ScoreBoard Updated”, “ScoreBoard Updated” to “Reset Round Score” and “Reset Round Score” to “Scores Reset”.
G. Start the system (game + simulation) and add tokens manually to “ScoreBoard Updated” and see if the score is incremented on the screen.

Task #5: Player Movement

A. Switch the current class net to “Player”.
B. Add three input places for the sensors “inputaxisvertical”, “inputaxishorizontal” and “mousex” and name them “Axis Vertical”, “Axis Horizontal” and “Can Rotate” respectively.

C. Add two transitions, both called “Or”, and link the places “Axis Vertical” and “Axis Horizontal”, each to a different “Or”.
D. Add a regular place called “Can Move” and link both “Or” to this place.
E. Add a transition “Move” that executes the script “Move” on the “player” channel and a place named “Moved”.
F. Link “Can Move” to “Move” and “Move” to “Moved”.
G. Add a transition “Rotate” that executes “Rotate” on the “player” channel and a place called “Rotated”.
H. Link “Can Rotate” to “Rotate” and “Rotate” to “Rotated”.
I. Start the system (game + simulation) and check if the player moves and rotates.

Task #6: Player Shooting

From now on, tasks will not have a step by step guide. Use the design doc for specifications on controls, sensors and adequate functions.

On the “Player” Class net, make a sequence that allows for the player to shoot balls. After you are done, test the system to see if the player shoots a ball when possible.

Task #7: Enemy Spawning

On the “HUD” Class net, create the enemy spawning mechanism. After checking the round winner, enemies should be spawned. Test the game by manually adding tokens to the place that will enable firing the transition responsible for spawning.

Task #8: Enemy AI

On the “Coward” Class net, make it so that actors seek when the see the player, scan surroundings when they don’t see him and shoot whenever it is possible. Start the system and check whether or not the enemies act this way.

Task #9: Scoring mechanism

Add to the “Coward” and “Player” a mechanism to update scores. This mechanism should work when the player is hit by an enemy and vice versa. After updating scores, enemies should die and the player should respawn. Test the game afterwards to see if this mechanism works as stated.

Task #10: Winner Announcement

On the “HUD” add a system that can announce a winner after the round score has been reset 8 times and play a sound when the winner is announced. Try out the game to verify if this works correctly either buy playing through the 7 rounds or by manually adding tokens to the affected places.

Task #11: Save for posterity

Save the project and name it after the number that was given to you at the start of the test.

Task #12: Time for interpretation

Import a project named “test.pnml” and explain how would an actor behave according to the presented petri-net.

D. Issue Compilation

Number	Issue	#	Type	Importance	Task ID	User ID	Fixed/Discarded	Solution
1	Did not know that objects are draggable.	1	A	Low	;4;	;8;		
2	Expected alphabetical order on lists.	9	A	Medium	;3;4;5;6;8;9;10	;3;4;6;7;8;9;10;11;	X	List were sorted alphabetically.
3	Expected that pausing the application would make it stop receiving tokens.	2	A	Medium	;3;	;3;10;		
4	Expected that the starting the simulation would run unity or vice versa.	1	A	Low	;3;	;5;		
5	Subnet did not show it could be unwrapped.	2	A	High	;1;12;	;2;5;		
6	Did not understand how Petri Net simulation works.	3	DK	High	;3;4;6;	;2;	X	Not related to application.
7	Did not understand the concept of sensor.	1	DK	High	;3;	;2;	X	Not related to application.
8	Did not understand the concept of weight.	4	DK	High	;3;10;	;2;6;7;11;	X	Not related to application.
9	Problem interpreting language semantics.	7	DK	High	;6;7;8;	;2;6;8;11;	X	Not related to application.
10	Problem interpreting Petri Net syntax.	18	DK	High	;1;3;5;7;8;9;10;12;	;1;2;3;4;6;7;8;9;11;	X	Not related to application.
11	Problem understanding the concept of receivers.	1	DK	High	;7;	;4;	X	Not related to application.
12	Buttons in tool bar are not fully sized.	2	FB	Low	;1;7;	;8;		
13	Did not find edit square present in the arcs.	3	FB	High	;3;	;1;4;9;	X	Increased the thickness and size of the edit square's border.
14	Did not know that numbers inside places corresponded to tokens.	1	FB	Medium	;3;	;1;		
15	Did not notice that place types are color coded.	1	FB	Low	;7;	;7;		
16	Did not notice he/she had the wrong cursor.	5	FB	Medium	;1;4;8;9;10;	;3;9;10;	X	Cursor's size was increased and there were added cursor changing toggle buttons.
17	Did not understand that he/she created a class net.	1	FB	Low	;1;	;10;		
18	Expected the dialogs to appear at center of window.	3	FB	Low	;4;5;8;	;3;5;9;	X	Made the dialogs appear where the click event was made.
19	Expected fusion places to be better indicated.	1	FB	High	;12;	;3;		
20	Got confused by Petri Net validator warning messages.	1	FB	Medium	;3;	;3;		
21	Ignored validator warning messages.	1	FB	Medium	;10;	;1;		
22	Arc did not disappear when changing cursors.	3	FE	Medium	;1;7;9;	;8;	X	See Issue 39.
23	Communication system stopped working.	6	FE	High	;7;9;10;	;7;8;9;11;	X	Added a better handler for when connections are ungracefully reset.
24	Editor crashed.	9	FE	High	;1;6;8;9;10;11;	;1;6;8;9;10;11;	X	Moved the message building call to another thread as to avoid a deadlock.
25	Editor viewport teleported after drag action.	1	FE	Low	;8;	;8;		
26	Graphical artifact.	3	FE	Low	;5;10;	;2;3;6;		

27	Mouse click was not registered properly.	39	FE	Low	;1;3;4;5;6;7;8;9;10;	;1;2;3;4;5;6;7;8;9;10;11;	X	Changed mouse event type from click to released.
28	Object teleported after undo.	5	FE	Low	;1;3;5;7;8;	;3;4;11;	X	Drag starting location coordinates were changed.
29	Play did not fire transitions.	5	FE	High	;4;7;10;	;4;5;8;9;11;	X	Made so that adding arcs would prompt the simulator to do another simulation step.
30	Validator not working.	2	FE	High	;3;4;	;5;8;		
31	Did not read crash course fully.	2	II	High	;1;12;	;2;	X	Not related to application.
32	Did not read design doc fully.	15	II	High	;6;7;8;9;10;	;1;3;5;6;7;8;9;10;11;	X	Not related to application.
33	Did not read task list fully.	9	II	High	;3;8;9;10;	;2;4;5;6;7;8;11;	X	Not related to application.
34	Problem interpreting crash course.	3	II	High	;1;12;	;3;7;8;	X	Not related to application.
35	Problem interpreting design doc.	4	II	High	;2;6;7;9;	;2;6;	X	Not related to application.
36	Problem interpreting task list.	20	II	High	;3;4;5;6;7;8;9;10;11;	;1;2;3;4;6;8;9;10;11;	X	Not related to application.
37	Problem understanding what to do.	2	II	High	;6;	;4;8;	X	Not related to application.
38	Tried to drag the cursor to create an arc.	13	M	Medium	;1;3;5;8;9;	;4;5;6;7;8;9;10;11;	X	Made so that dragging the cursor would create arcs.
39	Tried to drag from the add place button in order to create a place.	2	M	Low	;1;3;	;4;		
40	Used wheel scroll to move viewport.	1	M	Low	;3;	;4;		
41	Deleted class net by mistake.	3	M	High	;2;8;	;3;5;9;	X	Made removal of class nets to use the same method as other objects' removal.
42	Did not expect double clicks to remove elements.	1	M	Medium	;4;	;5;		
43	Did not find how to edit class net.	1	M	Medium	;2;	;2;		
44	Did not find where create class net button was.	6	M	High	;2;	;1;2;5;6;8;10;	X	Added "add class net" button to the tool bar.
45	Expected arcs to arrange themselves on object drag.	6	M	Low	;1;3;4;	;2;3;5;7;11;	X	Changed the arc anchor point and made the arc's curve control point increment when moving a linked object.
46	Expected a confirm dialog after new project.	1	M	High	;2;	;5;		
47	Expected copy paste.	5	M	Medium	;5;9;	;3;5;7;9;10;	X	Added copy paste.
48	Expected that pressing delete key would remove objects.	2	M	Low	;4;6;	;2;10;		
49	Expected dialogs to auto adjust their size.	3	M	Low	;4;8;	;7;11;		
50	Expected group selection.	4	M	Medium	;2;5;9;	;1;5;7;	X	Added Group selection.
51	Expected place dialog to have initial marking.	2	M	Medium	;4;8;	;3;7;		
52	Exported to the wrong net by accident.	1	M	High	;11;	;2;		
53	Mouse cursor hotspot was not calibrated.	7	M	Medium	;1;2;4;5;9;	;4;5;6;9;10;11;	X	Cursor hotspots were adapted according to the cursor's shape and size.
54	Pressed enter to confirm dialog.	31	M	Low	;2;3;4;5;6;7;8;9;10;	;1;2;3;5;6;7;8;9;10;11;	X	Allowed to confirm dialogs by pressing the Enter key.
55	Pressed ESC to cancel dialog.	5	M	Low	;1;3;4;5;8;	;1;4;11;	X	Allowed to cancel dialogs by pressing the ESC key.
56	Pressed wrong buttons on the tool bar.	4	M	Medium	;1;4;5;8;	;2;5;8;10;	X	Changed the order of the tool bar and added special separators.
57	Problem identifying how to change a place's type to input.	1	M	Medium	;3;	;1;		

58	Tried to double right click to edit objects.	2	M	Low	;3;5;	;8;		
59	Tried pressing ESC to undo special cursor.	4	M	Low	;2;4;5;6;	;1;2;4;	X	Made the ESC key a shortcut to undo special cursors.
60	Tried one click to change class net's name.	1	M	Low	;2;	;4;		
61	Tried one click to delete objects.	2	M	Medium	;4;	;2;4;		
62	Tried to cancel cursor by clicking outside of viewport.	3	M	Low	;1;5;9;	;2;3;6;	X	Added "default cursor" button.
63	Tried to click on object's name to edit its properties.	2	M	Medium	;3;6;	;2;5;	X	See Issue 64.
64	Tried to click on name to rename object.	7	M	Medium	;3;6;7;8;	;4;6;7;11;	X	Made it so clicks on name would open property dialogs.
65	Tried to click outside of dialogue to close it.	3	M	Low	;3;5;8;	;4;6;		
66	Tried to double click on arcs to edit their properties.	15	M	High	;1;3;5;6;8 ;10;	;1;2;3;4;5;6 ;7;8;9;10;1 1;	X	Introduced Arc property dialogs activated on double mouse click.
67	Tried to double click to add an object.	2	M	Low	;2;5;	;6;8;		
68	Tried to press ESC to remove objects.	1	M	Low	;1;	;2;		
69	Tried to press on object border expecting an action.	5	M	Medium	;1;3;	;3;4;6;8;11;	X	See Issue 39.
70	Tried to type with class selected in order to change its name.	1	M	Low	;2;	;6;		
71	Turned off the editor by mistake.	1	M	High	;8;	;2;		
72	Used right click to edit objects' properties.	1	M	Low	;2;	;7;		
73	Was confused on how to change to the default cursor.	2	M	Medium	;3;7;	;4;7;	X	See Issue 16.
74	Tried to drag a place to create an arc.	1	M	Low	;3;	;7;		
75	Confused because of output places.	1	NI	Medium	;9;	;1;		
76	Did not know channel == sensor.	7	NI	High	;3;9;	;3;4;5;6;7;9 ;	X	Changed name to sensor.
77	Did not know export == save.	10	NI	Medium	;4;11;	;1;2;4;5;6;7 ;9;10;11;	X	Export was changed to save.
78	Did not know receiver == channel.	6	NI	High	;3;9;	;3;4;6;7;9;	X	Change the "Receiver" label to "Actor ID".
79	Did not know regular == normal.	3	NI	Low	;3;5;	;4;7;8;	X	Not related to application.
80	Did not know script == invoke.	4	NI	High	;3;7;	;6;7;8;9;	X	Changed the "Invoke" label to "Script".
81	Did not know what "input" is.	2	NI	Medium	;3;8;	;11;		
82	Did not understand the language's naming.	4	NI	High	;1;2;3;	;2;3;4;7;	X	Changed "Transition" label to Action.
83	Did not understand what is the receiver.	5	DK	High	;7;8;9;	;1;6;7;8;11;	X	Not related to application.
84	Did not notice state of simulation.	7	PSS	Medium	;1;3;4;6;7 ;8;	;2;3;4;6;8;1 0;11;	X	Changed "Play" button to toggle button so that it is appears to be pressed when the simulation is running.
85	Expected if transition.	4	RI	Medium	;1;8;	;5;6;10;11;	X	Potential Future Work.
86	Expected OR transition.	1	RI	Medium	;5;	;10;	X	Potential Future Work.
87	Tried to stack transitions onto places .	2	RI	Low	;1;	;4;6;		
88	Tried to use transitions to add arcs.	2	RI	Low	;1;	;2;9;		
	Totals	406					50	

E. Raw data from Usability tests

Table 24 Demographic information of the test subjects

	Demographic Information					
Participant	Age	Sex	Qualification	TP	VP	GD
1	23	M	Msc Student	0	0	0
2	23	M	Msc Student	1	0	0
3	23	M	Bsc Student	2	0	1
4	36	M	Msc Student	1	1	1
5	28	M	Phd Student	2	0	1
6	22	M	Msc Student	0	0	0
7	29	M	Bsc	1	1	1
8	22	F	Msc Student	0	0	0
9	26	F	Msc Student	2	0	1
10	26	M	Bsc	2	1	1
11	24	M	Msc Student	2	2	2
Average	25,63	9M,2F	1 Bsc Student,2 Bsc, 7 Msc Students, 1 Phd Student	1,18	0,45	0,72

Table 25 Time per Task per User

	Users									
Tasks	1	2	3	5	6	7	8	9	10	11
1	05:31	05:51	04:35	05:56	07:45	02:27	05:44	06:03	07:43	05:31
2	07:41	07:16	06:40	08:15	10:42	09:14	06:36	05:27	04:38	05:29
3	08:52	13:25	15:25	12:16	10:12	13:25	11:43	11:10	05:49	06:46
4	06:25	07:42	09:19	07:11	06:07	10:18	10:19	05:31	04:39	06:36
5	06:40	09:48	08:56	07:32	11:07	11:30	09:14	07:37	06:33	08:54
6	03:45	08:59	04:40	05:09	10:00	07:45	15:22	04:34	06:26	07:03
7	05:03	08:13	06:05	04:13	08:14	10:43	13:24	05:15	04:02	06:44
8	08:05	09:06	08:44	07:56	11:22	13:05	17:02	22:48	09:02	11:24
9	14:55	13:54	21:21	07:40	12:04	22:02	14:26	12:48	08:18	12:13
10	08:32	06:20	06:53	03:03	05:43	11:39	10:26	08:46	03:28	08:32
11	01:02	00:40	00:40	00:23	00:44	01:08	00:44	00:59	00:38	00:33
12	07:23	09:49	05:41	03:08	02:09	04:37	05:03	04:03	02:29	02:19

Table 26 Difficulty per Task per User

	Participants									
Tasks	1	2	3	5	6	7	8	9	10	11
1	1	1	1	2*	3*	1*	1	2	2	3*
2	1	1*	1	1	2	2	2*	1	1	1
3	1*	2*	1*	2*	3*	2*	4*	4*	2	2*
4	1*	2	1*	2	3	2*	3	3	1	3
5	1	2	1	1	3*	2	3	2	1	2*
6	1	5*	1	1*	4*	2*	5*	2*	2	4
7	2*	4*	2*	2	4*	2*	5*	2	1	4*
8	1	3*	2*	2	4*	2	5	5	2	5
9	2*	2*	3*	1	5*	3*	4*	3	2	5*
10	2*	2*	3*	1*	4*	3*	4*	2	1	4*
11	1	1*	1	1	1	1	2	1	1	1
12	1	1	1	1	2	3*	5*	2	1	3

*= asked for evaluator's help.

F. User by Problem Matrix

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44
1										■			■	■							■			■			■					■				■								■
2					■	■	■	■	■	■																■	■				■		■		■	■							■	■
3		■	■							■						■		■	■	■						■	■	■				■		■		■					■			
4		■								■	■		■														■	■	■				■			■	■	■	■	■				
5				■	■													■									■		■	■		■	■					■			■	■		■
6		■						■	■	■														■		■	■					■	■		■	■		■						■
7		■						■		■					■								■				■					■	■	■				■						
8	■	■							■	■		■										■	■	■	■		■		■	■		■	■	■		■	■	■						■
9		■								■			■			■		■					■	■			■		■			■				■		■			■			
10		■	■													■	■							■			■					■				■		■						■
11		■						■	■	■													■	■			■	■	■			■	■			■		■						

	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88
1						■				■	■		■		■							■									■		■						■					
2	■			■				■		■		■			■		■	■	■			■		■			■						■					■		■				■
3	■		■				■			■								■				■			■							■		■				■		■				
4									■		■				■	■	■			■	■	■			■				■			■	■	■	■			■		■			■	
5	■	■	■			■			■	■		■							■			■										■	■								■			
6									■	■								■		■	■	■	■		■	■						■	■	■		■			■	■	■		■	
7	■		■		■	■	■			■										■		■						■	■	■		■	■	■	■	■		■	■					
8										■		■		■								■	■		■										■	■			■	■				
9			■						■	■												■										■	■	■		■								■
10			■	■					■	■		■										■											■							■	■	■		
11	■				■				■	■	■									■		■			■								■				■		■	■	■			

G.Task by Problem Matrix

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44
1					■					■		■				■	■					■		■			■	■			■			■				■	■					
2																																			■						■		■	■
3		■	■	■		■	■	■		■			■	■						■							■	■		■			■			■		■	■	■				
4	■	■				■										■		■									■		■	■						■						■		
5		■								■								■								■	■	■								■		■						
6		■				■			■															■			■					■			■	■	■							
7									■	■	■	■			■							■	■				■	■	■			■			■	■								
8		■							■	■						■		■						■	■		■	■				■	■			■		■			■			
9		■								■						■						■	■	■			■					■	■		■	■		■						
10								■		■						■					■		■	■		■	■		■			■	■			■								
11																								■												■								
12					■					■									■												■			■										

	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88
1	■								■		■	■						■				■		■	■													■		■	■		■	■
2		■				■			■	■					■	■							■			■		■										■						
3	■									■	■		■	■					■	■	■	■			■				■	■		■		■	■	■	■	■		■				
4	■			■	■		■		■	■	■	■			■		■																■							■				
5			■			■			■	■	■	■		■	■			■			■	■	■												■							■		
6				■						■					■				■	■		■																		■				
7										■										■									■							■			■	■				
8					■		■			■	■	■								■	■	■					■										■		■	■	■			
9			■			■			■	■								■													■	■		■					■					
10										■												■																						
11								■																									■											
12																																												

H. Priority leveled issues with color scaled values

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
0,000621983	0,537393641	0,004975867	0,000621983	0,014927601	0,016793551	0,00186595	0,059710405	0,156739812	2,418271384	0,00186595	0,002487934	0,016793551	0,001243967	0,000621983	0,093297507	0,000621983	0,016793551	0,00186595	0,001244	0,001244	0,0111957
23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44
0,13434841	0,60456785	0,00062198	0,011195701	2,40147783	0,0466488	0,13994626	0,0149276	0,007463801	1,2595163	0,4702194	0,033587103	0,059710405	3,02283923	0,007463801	0,646862716	0,002487934	0,000621983	0,033587103	0,001243967	0,001243967	0,067174205
45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66
0,055978504	0,00186595	0,062198338	0,004975867	0,007463801	0,044782803	0,009951734	0,00186595	0,26123302	1,735333632	0,046648754	0,079613873	0,001243967	0,002487934	0,029855202	0,000621983	0,004975867	0,016793551	0,009951734	0,139324277	0,011195701	1,84729064
67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88
0,004975867	0,000621983	0,062198338	0,000621983	0,00186595	0,000621983	0,009951734	0,000621983	0,001243967	0,156739812	0,223914017	0,111957009	0,011195701	0,059710405	0,004975867	0,089565607	0,139946261	0,365726228	0,039806936	0,001243967	0,002487934	0,002487934