Achieving reuse in casual game developers: A Blender Game Engine approach

Sven von Brand L.[1], Hernán Astudillo[1], and René Noel[2]

[1] Departamento de Informática, Universidad Técnica Federico Santa María,
Av. España 1680, Valparaíso, Chile
svbrand@inf.utfsm.cl
hernan@inf.utfsm.cl
http://www.inf.utfsm.cl

[2] Escuela de Ingeniería Civil Informática, Universidad de Valparaíso,
Valparaíso, Chile
rene.noel@uv.cl

Abstract. The Blender Game Engine (part of an open source 3D-modeling suite) offers a graphical logic editor interface, that's close to a node-based representation of game context behavior. The original design makes basic logic easy to make, but more complex logical solutions become hard to read, modify and reuse. This article describes a proposal for a visual programming node editor for the Blender Game Engine, by providing a Component Based System and building the node editor like a special-purpose, component-oriented system. A small, publicly available, proof of concept is made with basic functionality and tested by two external experts to validate the improvements that this work tries to achieve. With this proposal reusing and sharing generic logic between projects and teams is encouraged and modifying previous work products becomes much easier.

Keywords: Component oriented casual reuse visual programming

1 Introduction

Blender is an open source 3D creation suite[18]. It has great modeling, texturing and animation tools and it has many features that include physics simulation, post production, video editing and two internal rendering engines. Blender has a working visual interface to make interactive content; this part of Blender is called the Blender Game Engine. The engine has support for physics, excellent OpenGL features, access to a working animation system, a Python API and a graphical logic editor to create the interactive content. The current logic editor can be combined with Python to make advanced interactive content, but the visual logic made with the engine is considered hard to re-use and hard to read.

During this work a new system is proposed, that it is expected to solve some of the problems and to extend some functionality, making visual logic easier to reuse, share and modify. The implementation of the new system is explained, for a validation study to be done afterwards. The study is comprised of an expert opinion on the new system and a small experiment with university students testing the system.

Through the analisys and implementation of a new visual programming tool for blender we propose an approach that helps solve the reuse problem in visual programming enviroments, through design choices that enable and push users to reuse in their projects.

2 Context

It is common in the video game developing industry to use game engines, which provide many commonly used functionality readily available, such as sprites, 3D rendering, physics, texturing, animation systems and more. Game engines are used by small and big teams alike, but there are different trends among small and big teams. Platform-specific game engines are used in the gaming industry, but most developers use multi-platform engines such as Unity 3D, Unreal Engine and Cry Engine.

Game engines can in general be divided into 2D and 3D engines, being Blender in the second group. This engines offer different programming languages as options and some have their own programming language. Blender, in particular, is compatible with Python. The engines usually provide an API to access the functionality it provides. API are engine specific and many engines have support for more than one programming language.

Many of the popular game engines use a visual interface that allows to manipulate environments, create objects and define global effects, such as physics and lighting. This interface also allows to add code and execute the game, or portions of the game called scenes or levels. Engines manage the deployment of the game to different platforms and usually has debugging tools included.

Some game engines have visual programming tools for some features, such as animation diagrams, state machines and dialog trees. There's also some engines that have 3rd party visual programming tools, like Unity. Construct 2 [17], a 2D professional engine, has a Visual Programming Language (VPL), that's very close to code written in JavaScript as one of it's main features.

Blender allows the use of most of its graphical features through its game engine, but to make larger games, Python is needed. The best example of a game made with the Blender Game Engine is "Yo Frankie!" [16], a game made by the Blender Foundation to show the game engine's capabilities. The current graphical logic editor (see figure 1) is based in three columns of elements:

- Sensors: This elements check for a certain action to be met, in which case they through either a true if the condition is met or a false if a action is not met. Examples are keyboard events, property comparisons and collision detection.
- Controllers: This elements are logic statements which will send a signal to any connected actuator if they are met. This can be a preset of logic statements or a defined expression. It is also possible to run a Python script instead of using a logic statement. This script can access all connected Sensors and Actuators and can also do some actions directly through the Blender Game Engine Python API. The Controllers can be associated with a state, this functionality allows to be able to encapsulate some logic and to make conditional and triggered behaviors in objects. States are exclusive to controllers.
- Actuators: This elements are triggered when they receive a signal from a controller, this are visible actions in the content like motion, editing the object mesh and changing properties values.

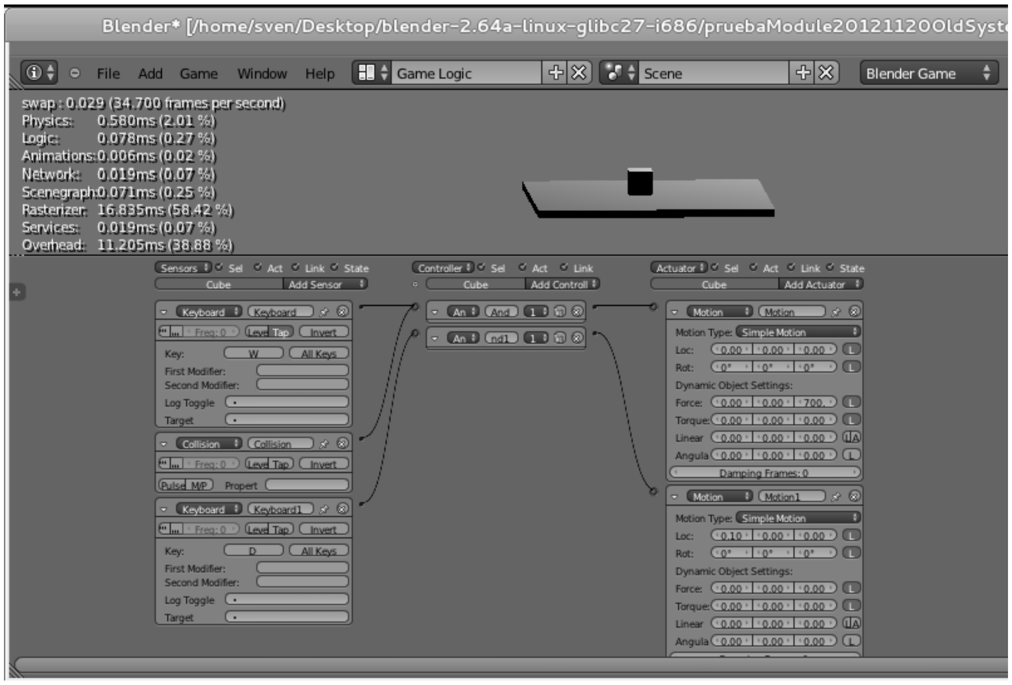
Currently the system checks sensors and controllers in each object to make each logic step, if an object is duplicated this mean that all the needed data for the object must be duplicated also. A game of a simple platform mechanic is made to show the basic functionality. A Cube that can jump and move forward is shown in figure 1. The 'w' key allows to jump, only when the object is colliding with something and the 'd' key allows to move forward at any moment.

3 The Reuse Problem

Even though Python is used for bigger games, it is always necessary to combine Python with the graphical logic editor present in Blender. Without the graphical logic editor it is not possible to access many features.

Blender's three-column graphical logic editor results are integrated directly into one blender game objects when used, unlike other features of blender and the logic in other modern game engines, that are more modular and independent of the objects. Blender game objects work as a reference attached to each element in the columns and what should be a reusable component, is currently a specific game object element. This makes it very hard to actually reuse large portions of logic and to read the behavior of logic.

As more logic elements are added to a game object, the current system makes the visual logic representation to grow down, without the possibility of encapsulating logic beyond the



Blender* [/home/sven/Desktop/blender-2.64a-linux-glibc27-i686/pruebaModule20121120OldSyste

```
swap : 0.029 (34.700 frames per second)
Physics:     0.580ms (2.01 %)
Logic:       0.078ms (0.27 %)
Animations:  0.006ms (0.02 %)
Network:     0.019ms (0.07 %)
Scenegraph:  0.071ms (0.25 %)
Rasterizer:  16.835ms (58.42 %)
Services:    0.019ms (0.07 %)
Overhead:    11.205ms (38.88 %)
```

Keyboard: Keyboard — Freq: 0, Level Tap, Invert — Key: W, All Keys — First Modifier: — Second Modifier: — Log Toggle — Target

Collision: Collision — Freq: 0, Level Tap, Invert — Pulse M/P, Propert

Keyboard: Keyboard1 — Freq: 0, Level Tap, Invert — Key: D, All Keys — First Modifier: — Second Modifier: — Log Toggle — Target

An: And, 1

An: nd1, 1

Motion: Motion — Motion Type: Simple Motion — Loc: 0.00, 0.00, 0.00 L — Rot: 0°, 0°, 0° L — Dynamic Object Settings: — Force: 0.00, 0.00, 700. L — Torque: 0.00, 0.00, 0.00 L — Linear 0.00, 0.00, 0.00 L A — Angula 0.00, 0.00, 0.00 L — Damping Frames: 0

Motion: Motion1 — Motion Type: Simple Motion — Loc: 0.10, 0.00, 0.00 L — Rot: 0°, 0°, 0° L — Dynamic Object Settings: — Force: 0.00, 0.00, 0.00 L — Torque: 0.00, 0.00, 0.00 L — Linear 0.00, 0.00, 0.00 L A — Angula 0.00, 0.00, 0.00 L

Fig. 1: Current Blender Game Engine Graphical Logic Editor

use of states. States allow to encapsulate logic, but it is not possible to save state's logic independent of a game object.

With Python it is possible to do reusable scripts, but this needs to be implemented in combination with the current visual logic to access some features of the Blender Game Engine. Blender has many functionality that are accessed through its interface, this include defining physics, materials, lighting and initial model positions. Most of this functionality are also set for the Blender Game Engine, making it unnecessary to program initial object positions, materials or the physics. For non-programmers it is very hard to do a clean project, specially when many game-object have similar behaviors. This makes bigger project rely in Python in a more direct way, but Python scripts are hard to reuse, as they must be related to specific visual programming elements.

The main problems identified, that need to be solved are:

– Visual logic is hard to comprehend, follow and modify
– Logic is not easily reusable, easy to combine or to share
– Engine must keep current capabilities with the new system

4 Existing Approaches

There are many implementations aimed at making development possible through visual interactions by non-programmers.

Developed at MIT, Scratch is a Visual Programming Environment(VPE) with three core design principles, expressed in Scratch: programming for all [6]: *"Three core design principles for Scratch: Make it more tinkerable, more meaningful, and more social than other programming environments."*

In Scratch [6,7], it is described as a visual programming environment that allows people to create different kind of projects, such as animated stories, games, music videos, science project and other multimedia applications.

The grammar of Scratch is related to Lego Mindstorms[13], aimed to make visual development restricted and more intuitive. An important feature of Scratch is the possibility to easily share and modify projects. Scratch's openness allowed for a very big community to form around the creation of content.

Unfortunately Scratch is a 2D engine completely integrated and isn't designed for it's components to be used in other scopes. It is also very different to how Blender manages it's logic and would be very hard to implement technically and for the users to learn.

There are many industry approaches to the same problem. A common target demographic for professional tools are the game designers that don't have a technical background, as expressed in eCo: Managing a Library of Reusable Behaviours[12]

> *A common approach to help designers is to let them use visual languages that are supposed to facilitate the process by hiding the formal syntax of the underlying programming language. UnrealKismet, integrated in the Unreal Development Kit game editor, and Flow-Graph Editor, integrated in the Sandbox Editor of CryENGINE 3 SDK, are of two such visual scripting tools that let designers model the gameplay of a level without touching a single line of code through some variation of data flow diagrams. Also, the celebrated Unity 3D has plugins like Behave or Playmaker...*

The eCo tool is designed to allow game designers to use behaviors with an emphasis in having developers add the more complex elements through code, and giving a high level tool to game designers. Even though eCo is aimed at reuse and sharing, it is not aimed at giving a complete set of tools to make games to designers, but as a high level tool to connect developers and designers.

5 A Visual Component-Oriented Editor

Visual component-oriented systems provide a novel way to address the reusability problem, through the design of a visual programming language; this also implies elaborating a formal language definition and providing examples to clarify how it works. The proposed language must be focused in a broad audience, because of this, the ease of use and ease to learn are the most important features the language must have. Although the domain treated in this work is specific to the Blender Game Engine, the domain should be generic enough to be reused in other systems, that contain game objects equivalent to those present in the Blender Game Engine. It is proposed to make the system integrated into Blender through an Entity System.

There are some discussions about how an Entity System should be made or exactly what does it mean, but for the work currently done the definition that best describe the work is that of Adam Martin in his website [2].

> *What's an Entity?*
>
> *An Entity System is so named because Entities are the fundamental conceptual building block of your system, with each entity representing a different concrete in-game object. For every discernible "thing" in your game-world, you have one Entity. Entities have no data and no methods.*
>
> *In terms of number of instances at runtime, they are like Objects from OOP (if you have 100 identical tanks, you have 100 Entities, not 1).*
>
> *However, in terms of behaviour, they are like classes (Entities indirectly define all the behaviour of the in-game object - we'll see how in a second).*
>
> *What's a Component?*
>
> *Every in-game item has multiple facets, or aspects, that explain what it is and how it interacts with the world.*
>
> *(...)*
>
> *The way that an Entity for an item represents the different aspects of the item is to have one Component for each aspect. The Component does one really important thing:*
>
> *Labels the Entity as possessing this particular aspect*

An Entity System is a system that's based on the proposition that there are entities which can add, remove and use different components to define its behavior and properties. The entity contains components dynamically, while methods and properties are exclusive to components. This means that an entity is a combination of instantiated components. As such it is recommended to keep the entities as a very clean element in the system, allowing to easily change the behaviors of entities with ease and there would be less code replication as there's the clear ability to reuse components easily and even in runtime.

The components of the system must be self contained to allow them to work with many different entities and keep the flexibility of the system as high as possible.

As for the node editor, the definition of it as a programming language allows to solve many problems by using the body of knowledge of programming language engineering. In this aspect, the current work follows very closely Anneke Kleppe's work in her book "Software Language Engineering, Creating Domain-Specific Languages Using Metamodels" [1]. It might sound strange to some to treat a visual representation as a programming language and one might think the node editor is translated to some form of code that is the actual programming language, but the powerful idea behind this project is to actually treat the node graphs and visual representation as the language itself.

The functionality that makes the Blender Game Engine attractive and must be kept are:

- Mixing sensor, controller and actuator elements related to different game objects to make a compound.
- Runtime game object duplication also duplicates relations between game objects. This is currently done by duplicating sensor, controller and actuator elements and their relation with other game object's elements, the implementation might be changed in the new system, but the functionality must be kept.
- Easy access to Blender's functionality, such as animations, materials, mesh objects, boned animations and lighting.

The initial proposed node graph visual programming language is a very simple one which has only the following base elements:

1. Entity: The representation of a group of functioning and defined components. It can be related to one or many components, it doesn't have any data, only a relation with labels and local values for components related to it.
2. Label: An identifier represented by a string, that allows to relate one or more entities to one or more components.
3. Component: The combination of logic as graphs and data as properties. It can be related to entities through labels.
4. Graph: The functional combination of nodes into logic.
5. State: A string label in a component that activates or deactivates specific graphs inside a Component.
6. Node: Receives and sends information through links, which are connected to Data-in and Data-out pins. Has internal logic to handle information.
7. Push-in and Push-out pins: Pins are related to links, each Pin can be connected to multiple links, but only be part of one Node.
8. Pull-in and Pull-out pins: Pins are related to links, each Pin can be connected to multiple links, but only be part of one Node.
9. Push Links: A Push Link is related to a Push-out pin and a Push-in pin. Push Links have internal logic to delay transporting data and filtering, it transports entity lists from one node to another to activate them.
10. Pull Links: A Pull Link is related to a Pull-in pin and a Pull-out pin. The link works on demand and doesn't have logic attached to it.
11. Properties: Properties are specific to Components. A property can only be related to one component and accessed through a node in the graph. Properties can be private to the component, public to all components or available only to components related directly to the same entity. Also properties can be at component level or at entity level, being the later the default use of properties. At component level, a property will only have two scopes, private to the component and global.

The language will be described with an abstract syntax and a proof of concept. In figure 2, the rules of relations between the components is described, but it doesn't give information about how they should be combined to make a working program. This abstract syntax is enough as a base of rules to follow, to make a simple syntax checker and to have an idea of how a program should be modeled using this language.

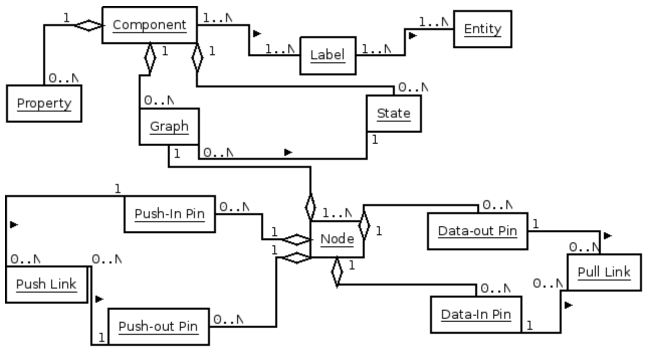


Fig. 2: Simplified Abstract Syntax

6 Implementation

A small graph editor was made [18], that replaces Blender's three-column logic editor. This editor can compile graphs into working behaviors represented in Python code. Finally this code is loaded through a special program runs inside of Blender, that assigns behaviors to game objects by matching a property name with a behavior name. The logic being edited is generic and not related to a specific game object, a game object can have more than one behavior associated to it and many objects can use the same behavior.

The graph editor has the functionality of adding nodes, connecting nodes, changing the type of the nodes, moving and removing nodes, removing node connections, saving and opening graphs and finally compiling a graph into a working behavior.

States, labels and pull pins weren't implemented in the proof of concept.

The type of node created for this proof of concept were:

1. Move Node: Makes the object move in its local coordinates each frame in the value given, then passes to the next node.
2. Rotate Node: Makes the object move in its local coordinates each frame in the value given, then passes to the next node.
3. Keyboard Node: Passes to the next node only if the Key is in the state selected.
4. Action Node: Allows to play an animation, then it passes to the next node.
5. If Node: Evaluates a logical condition, it has two output pins, at False, it passes through the '0' pin, at True, it passes through the '1' pin.
6. Set Property: Allows to manipulate a game object property, then it passes to the next node.
7. Add Object: Allows to add a new object to the current scene, then passes to the next node.
8. Collides: Checks if the object is colliding with the ground. This is an very specific node, as the colliding information is not currently available through Python in Blender. If it collides it continuous through the pin labeled "1", if it doesn't it continuous through the pin labeled "0". It only works if the object's physics are enabled in Blender.
9. Set Velocity: Sets the velocity for the object, it only works if the object's physics are enabled in Blender.

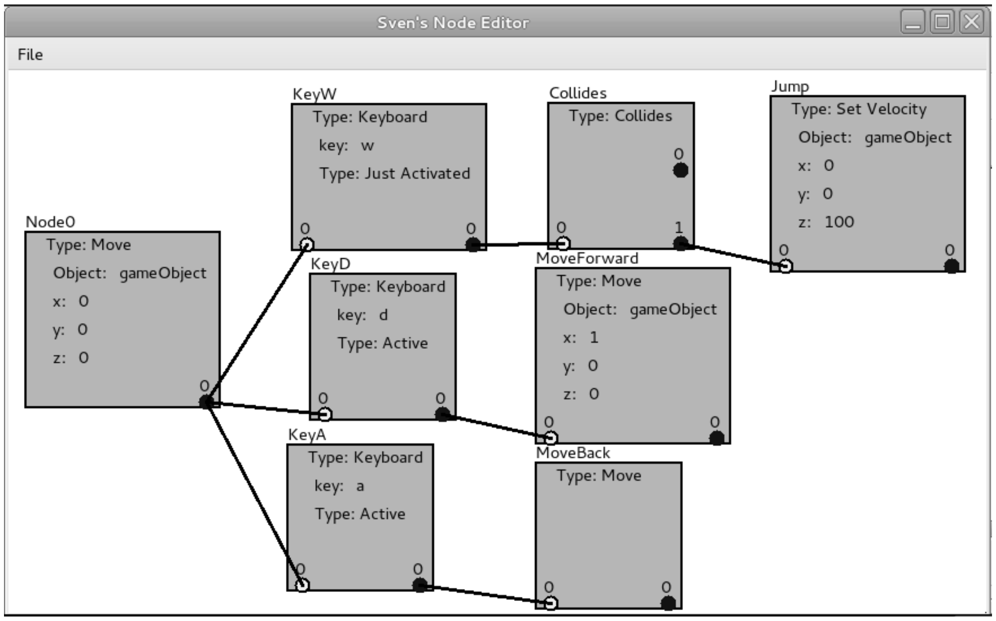
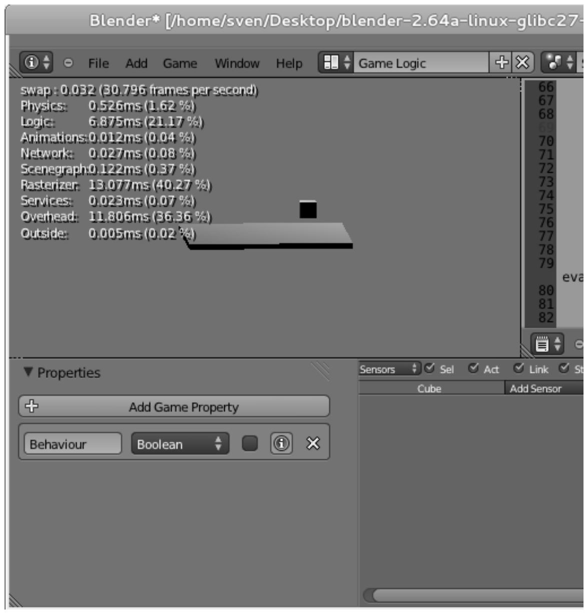


Fig. 3: Graph editor proof of concept



```
swap : 0.032 (30.796 frames per second)
Physics:      0.526ms (1.62 %)
Logic:        6.875ms (21.17 %)
Animations:0.012ms (0.04 %)
Network:      0.027ms (0.08 %)
Scenegraph:0.122ms (0.37 %)
Rasterizer:   13.077ms (40.27 %)
Services:     0.023ms (0.07 %)
Overhead:     11.806ms (36.36 %)
Outside:      0.005ms (0.02 %)
```

Fig. 4: Adding a behaviour to the cube in Blender

In figure 3 a simple behavior example is made through nodes. The “Node 0” is called at the start of every frame and it will automatically call all connected nodes. Keyboard nodes only call the next node if the key is in the state selected. Inside of Blender, a small program is set to run every frame to run behaviors created. The default cube is selected and by adding a property with the name of the behavior, the Cube can be moved using the keys ‘w’, ‘d’ and ‘s’. It must be noted that one game object can have multiple behaviors at the same time and multiple objects can have the same behavior.

The graph is compiled as "Behaviour.py" file inside a specific directory. The name of the behavior must be added as a property inside of Blender, so that the runtime program can call it. In figure 4 the property that relates the cube to the behavior and the game running can be seen inside of Blender's interface.

This is a top level design that describes the user elements available to make programs and tries to give an inside on the strengths and functionality that make it an improvement over the current system. This proposal doesn't go into detail on technical implementation as it is focused on the user and not the underlying system, this means that the design is flexible enough to be implemented on other software. As this is a top level proposal for a new functionality in Blender, it will be needed to test it with users and it is proposed to do a Top-Down development. It is also proposed to make an iterative prototype driven implementation, making prototypes that allow to test the power and flexibility of the proposed functionality giving priority to the ease to change and expand prototypes through user needs and testing results, modifying the language upon finding problems or possible improvements and changing the prototype to meet the language requirements. Upon getting to a stage where the language is well defined and the prototype proves to be powerful enough, it is planned to optimize the prototype or make a more optimized implementation based on the prototype.

The current proof of concept was implemented using Python. It has two programs that are needed for any example to run:

1. The graph editor: This editor allows to make and compile graphs into behaviors. It was implemented using wxPython [18].
2. A Blender runtime module: This module instantiates behaviors, relates them to game objects and runs the behaviors every frame. It also helps manage the portal nodes. It was made using Python using Blender.

All the images are taken from the programs running from a machine running Fedora Linux.

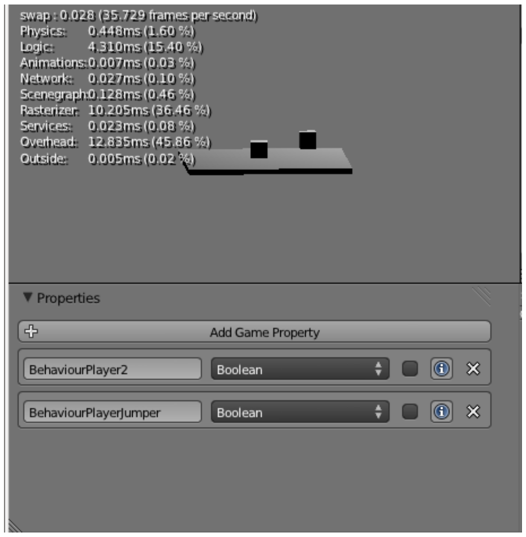
Another example was done to show the flexibility of the new system. For this the portal nodes presented in are used.

First the behavior of moving around and jumping is done with portal in nodes, as it can be seen in figure 5. Then another behavior is done, that calls the portal in nodes upon defined key presses. This graph is then compiled into a behavior called "Behaviour-PlayerJumper.py". The first cube controller graph is then compiled to the behavior called "BehaviourPlayer1.py". The controls for this cube are 'w' to jump, 'd' to move right and 'a' to move left. An equivalent graph is made for the second cube, but the keyboard nodes are changed to the up arrow, right arrow and left arrow.

Finally inside of Blender the first cube is given the properties "BehaviourPlayer1" and "BehaviourPlayerJumper", while the second cube is given the properties "BehaviourPlayer2" and "BehaviourPlayerJumper". This allows to easily modify and extend both player's behaviors without the need of making two separate "BehaviourPlayerJumper" behaviors, while keeping the controls separated. Comparing this, to the previous system, having two symmetrical controlled game object meant that each time one had to change a value, for example the velocity of the jump or the amount of movement, one had to change it manually in each object. Also if this was extended to the behavior of many different elements in the game world, it meant a lot of work, or the use of Python. Making changes manually through many objects usually results in errors and the use of Python meant that non-programmers were alienated of more simple solutions. In the example made with the proof of concept, changing the velocity of the jump will automatically change it to both players, without affecting their controllers. Also a third cube, with different controllers is easy to introduce, by making only a new controller behavior for it and using the "BehaviorPlayerJumping.py" component.

7 Validation Through Expert Opinion

The proposal was validated in two manners, the first, by the expert opinion of two advanced professional users of Blender.



```
swap : 0.028 (35.729 frames per second)
Physics:      0.448ms (1.60 %)
Logic:        4.310ms (15.40 %)
Animations:0.007ms (0.03 %)
Network:      0.027ms (0.10 %)
Scenegraph:0.128ms (0.46 %)
Rasterizer:  10.205ms (36.46 %)
Services:     0.023ms (0.08 %)
Overhead:    12.835ms (45.86 %)
Outside:      0.005ms (0.02 %)
```

BehaviourPlayer2 Boolean

BehaviourPlayerJumper Boolean

Fig. 5: Cube 1 properties and game running

The proof of concept was presented to two professional advanced users of Blender, Mowgly Schwarzwildhirsch and Cristián Brevis. Schwarzwildhirsch is an animator and a multimedia developer that has used Blender for ten years and Brevis is a System Analyst and an Audiovisual Technician that has used Blender for two years. Both professionals were shown how the system worked, and allowed to make a simple example. By just explaining the controls of the graph editor, one of them did a simple test of a keyboard triggered event.

The example was being developed through a shared folder and it was possible to see and test each others graphs very fast, as the only integration a behavior needs, when being compiled, is to be added to a specific folder of the project. This was designed like a common Java library structure in mind, to allow the importing and sharing of behaviors, like it is done with Java libraries. After Brevis did this example, he run it in Blender, by adding the property to the base cube. The .blend used by Brevis was previously modified to use the main loop Python program that allows the execution of behaviors. This meant that the example run by just adding the property. Then Brevis and Schwarzwildhirsch were given the proof of concept to test for themselves and a set of questions was given to them.

After the experience both experts were asked to give their opinions about the new implementation's performance, maintainability, efficiency and usability. On the subject of maintainability, efficiency and team work both agreed that the proof of concept presented clear advantages, being the library like structure and the component oriented approach the main reason cited for this. On the subject of usability, Brevis proposed to make the interface closer to Blender's and also said that there weren't enough hints in the graph editor. He also said it would be useful to have a way of testing the behavior, once the graph is done, without needing to do the whole process of compiling and assigning. Schwarzwildhirsch said he thinks that, once the system is complete, it would be easier to use, but that in it current state it was hard to asses.

8 Validation Through Experiment

A small activity was made with fourteen students in a course of game development. All of the students had basic knowledge about Unity, SVN and more advanced knowledge about programming languages. The course was divided in two groups, people in each group was asked to complete three simple tasks. The course is comprised of students from different levels and work in small teams of 2 or 3 persons. To have comparable groups in the experiment, each team had members in each experiment group. Group 1 used Blender and group

2 used Blender with the graph editor presented in this paper. Before starting the experiment students were shown the basic functions of the new system by showing the examples presented in this work and were allowed to make questions only about the interface during the experience.

The first task was to fix an implementation that had a game object moving with simple controls, but the controls were inverted. The second task was to combine the controls of two separate game objects into both of the game object and then make some modifications to the controls. And the final task, was to implement the same behaviors in a project with different graphical assets, with the freedom to open and reuse everything used so far.

For each of the three experiences performed by the groups, we collected both quantitative and qualitative data. We measured the time required to finish the activity, and also if it was successfully completed. We collected subject's opinion about the activity, and an expert Blender developer commented the results.

During data collection, results were examined in order to check if the subjects seriously performed the activity. One of the subjects did not finish any activity, and according to his comments, he didn't understood what to do. It was the only case, so this data point was discarded. Another abnormal situation was observed in the completion time for the experience 3, a subject spent 15 minutes with configuration issues, and his final time for the experience 3 was 20 minutes. This data point was modified to 5 minutes, in order to reflect real activity effort.

Following table summarizes qualitative data collected, which will be analyzed in the following sections.

Table 1: Collected Data, where group 0=subjects using Blender and 1=Subjects using SvenNodes

		Experience 1		Experience 2		Experience 4	
Id	Group	Time	Complete	Time	Complete	Time	Complete
1	0	4	1	10	1	15	1
2	0	3	1	10	1	15	1
3	0	1	1	10	1	3	1
4	0	1	1	10	0	10	1
5	0	8	1	10	0	15	1
6	1	4	1	5	1	5	1
7	1	5	1	4	1	2	1
8	1	7	1	5	1	10	1
9	1	7	1	4	1	2	1
10*	1		0		0		0
11	1	10	1	10	0	15	1
12	1	5	1	8	1	5	1
13	1	3	1	8	1	15	1
14	1	10	1	10	0	18	1

8.1 Data Analysis

After collecting the data, a descriptive analysis was performed. We defined three variables to test the hypotheses: the total completion time for each subject (avTime), total completion time (totalTime) and number of completed experiences (QoE).

First, we explored descriptive statistics for the collected data in order to look for outliers and distributions for the variables avTime, totalTime and QoE. Data was separated according to the experimental groups. Figures 6, 7 and 8 show box plots for each variable.

No outliers were found. The medians for each variable and group are presented in the table below.

Normality test allow us to perform a parametric test on time variables, but a non parametric approach must be used for QoE variable.su

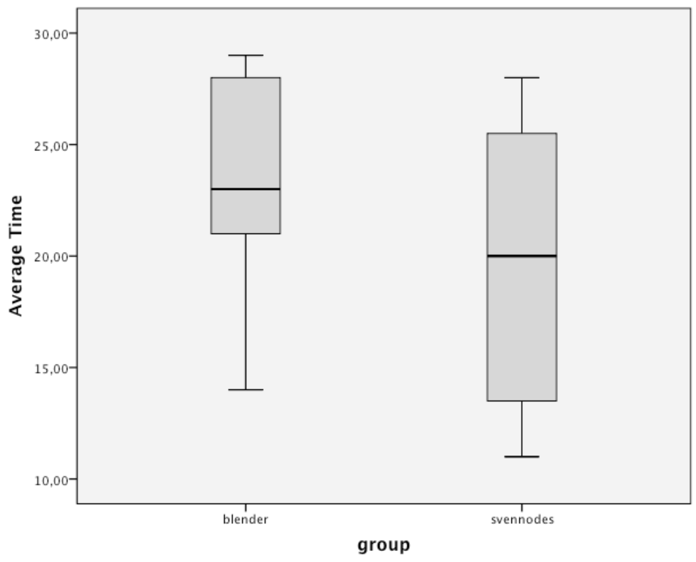


Fig. 6: boxplot for avTime variable.

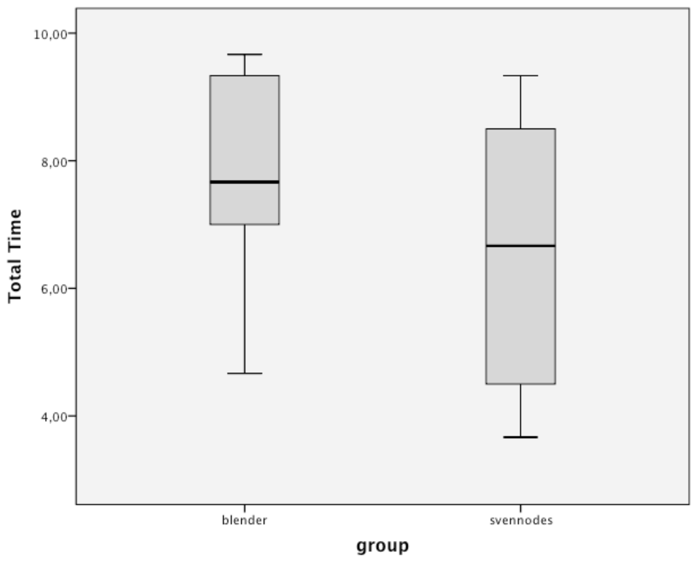


Fig. 7: boxplot for totalTime variable.

8.2 Hipothesis Testing

We performed a One-way ANOVA for avTime and totalTime variables. Homogeneity test for variances show that this assumption is not violated (p=0,478 for both variables).

ANOVA results for avTime and totalTime are not statistically significant (p=0,478). So, no significant differences in time can be stated.

For QoE variable, we performed a Kruskall-Wallis H test. No statistically significant differences were found (p=0,584). A non-parametic test was done for QoE because Shapiro Wilk test showed that the distribution of both groups results were not normal.

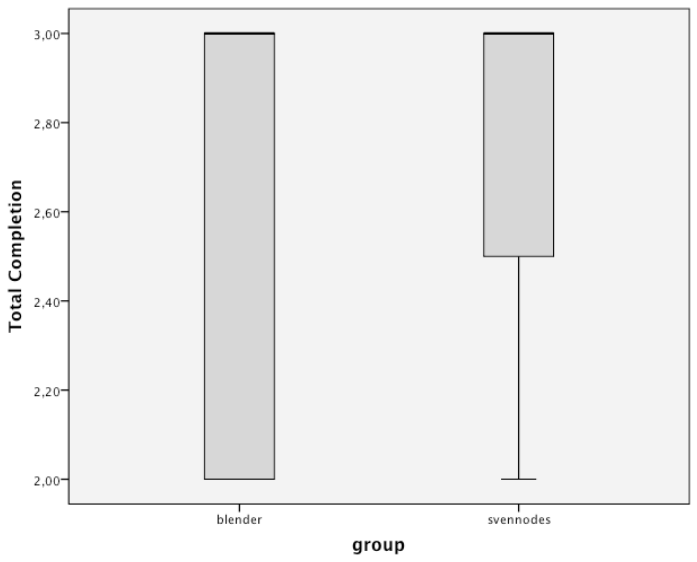


Fig. 8: boxplot for QoE variable.

Table 2: Descriptive statistics and Normality test for avTime, totalTime and QoE

Group		Average Time	Total Time	Total Completion
Blender	Media	23,000	7,6667	2,600
	N	5	5	5
	Desv. tip.	6,04152	2,91384	,54772
svennodes	Media	19,6250	6,5417	2,75000
	N	8	8	8
	Desv. tip.	6,52331	2,17444	,46291
Total	Media	20,9231	6,9744	2,6923
	N	13	13	13
	Desv. tip.	6,31746	2,10582	,48038

8.3 Validity Analysis and Results Discussion

Main validity threats in this first experimental design with exploratory purposes were related to internal validity. Time restrictions forced us to take an time-box approach to perform each experience, so subjects that didn't complete the activities were rated wit the maximum time defined for the activity.

ANOVA results for avTime and totalTime are not statistically significant (p=0,478). So, no significant differences in time can be stated.

8.4 Quality of solutions

The group using the new tool had more members having the approach of reusing elements and in the group using Blender with no modifications, one of the testers complained about not knowing how to easily reuse the logic and all testers said they just did everything manually each time.

All testers using the new tool tried to reuse elements, one of them could not because of a technical problem and another tester was trying to make everything from scratch, but when he got an error, he decided to reuse instead.

Table 3: Descriptive statistics and Normality test for avTime, totalTime and QoE

		Kolmogorov-Smirnov			Shapiro-Wilk		
	group	Estadístico	gl	Sig.	Estadístico	gl	Sig.
Average Time	Blender	,196	5	,2	,931	5	,604
	svennodes	,181	8	,2	,920	8	,426
Total Time	Blender	,196	5	,2	,931	5	,604
	svennodes	,181	8	,2	,920	8	,426
Total Completion	Blender	,367	5	,026	,684	5	**,006**
	svennodes	,455	8	,000	,566	8	**,000**

Table 4: test of homogeneity of variances

	Estadístico de Levene	gl1	gl2	Sig.
Average Time	,541	1	11	,478
Total Time	,541	1	11	,478

8.5 Summary

The experiment showed that with no extra effort, testers in the second group made better solutions in terms of reuse. A very important aspect of the experiment is that when a user tried to reuse in the old system, the system did not allow him to do this in a simple manner, but in the case of the new system one of the users tried to redo everything and was ultimately convinced to reuse.

9 Conclusion

The current system present in Blender does not encourage tweaking logic applied to a group of objects, as it is common to have to apply changes manually to each object. There are ways to artificially make local logic of game objects more manageable, but the proposed solution would encourage to do reusable logic to beginners and make team work more manageable with visual programming, by presenting a standard approach.

The proposed node-based syntax for logic has many similarities to the current game engine, but expands it, by giving the ability to have explicit larger logic sequences and more options to the user to make logic more readable and modifiable.

Both the proof of concept using Python and the current system are strong indicatives that a system like this would expand the current engine's functionality in the fields of reusability, sharing, visualization of logic and ease to modify logic.

The proof of concept showed that it is viable to do a component oriented approach, but there are several limitations that don't allow to implement all of Blender Game Engine's features through Python, which means that to fully implement the system, low level modification must be done to the Blender. The independent nature of the graph editor implemented for the Proof of Concept would allow to apply it to other engines.

The experiment showed, that even though the new system signified more effort, by using a separate graph editor, testers used the same time and got a better result in terms of reuse.

One of the problems with the new approach against the current is that, as the design is to keep component modular and self contained, it is a little bit harder to directly relate the behavior of two objects in the game.

The proposed solution allows the reuse of visual code and it is designed to be easy to read and understand. The reuse of components with an entity system design allows to easily import a component from one project to another and reuse it with very little work, allowing this in the future to manage bigger projects and share solutions with the community much easier. This is made possible by following the idea that a component must be self contained and that it cannot depend upon a certain kind of entity for it to work.

The most notable advantage in designing a game is the possibility to easily modify the behavior of entities, relating components with entities by applying label changes, specially because it allows explicit logic reuse and the system design pushes the developer into making the logic reusable.

The work presented suggests that to give visual programming environments new strengths requires new design approaches to the implementation of the tools. the design should take in consideration how to enable the use of good practices and available functionality used by the industry. New problems to be explored in this field include versioning control and debugging.

For future work it is expected to be able to conduct experiments with higher confidence in the results and isolating the impact of the proposed solution in the experiment's results, also by having the proof of concept implementations in a more stable state or having a stable prototype.

Acknowledgments. This work was partially supported by UTFSM (grants DGIP 24.12.50) and CONICYT (CCTVal Basal FB0821).

References

1. Kleppe, Anneke. "Software Language Engineering: Creating Domain-Specific Languages Using Metamodels". Adison Wesley, United States 2008
2. Martin, Adam. http://t-machine.org/ T-Machine Blog posts on Entity Systems. 2007 through 2010. http://t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/
3. Blender Foundation. Hierarchical Nodal Logic for Blender 2.5. 2009 through 2010. http://www.ncbi.nlm.nih.govhttp://wiki.blender.org/index.php?title=Dev:Source/GameEngine/NodalLogic
4. Ju An Wang, Andy and Qian, Kai. "Component Oriented Programming". Wiley Interscience, Hoboken, New Jersey , United States 2005
5. MacLaurin , Matthew. "The Design of Kodu: A Tiny Visual Programming Language for Children on the Xbox 360" . ACM SIGPLAN Notices - POPL '11 - Volume 46 Issue 1, January 2011, Pages 241-246.
6. Resnick, Mitchel; Maloney, John; Monroy- Hernández, Andrés; Rusk, Natalie; Eastmond, Evelyn; Brennan, Karen; Millner, Amon; Rosenbaum, Eric; Silver, Jay; Silverman, Brian and Kafai, Yasmin . "Scratch: Programming for All" . Communications of the ACM - Volume 52 Issue 11, November 2009, Pages 60-67.
7. Maloney, John; Resnick, Mitchel; Rusk, Natalie; Silverman, Brian and Eastmond, Evelyn. "The Scratch Programming Language and Environment". ACM Transactions on Computing Education (TOCE) - Volume 10 Issue 4, November 2010, Article No. 16.
8. Cooper, Stephen. "The Design of Alice". ACM Transactions on Computing Education (TOCE) archive - Volume 10 Issue 4, November 2010, Article No. 15.
9. Cooper, Stephen, Dann, Wanda, and Pausch, Randy. "Alice: a 3D Tool for Introductory Programming Concepts". In Proceedings of the Fifth Annual Consortium for Computing Sciences in Colleges - Northeastern Region (CCSCNE), Ramapo, New Jersey, USA, 2000, Pages 107-116.
10. Kölling, Michael. "The Greenfoot Programming Environment". ACM Transactions on Computing Education (TOCE) archive - Volume 10 Issue 4, November 2010, Article No. 14.
11. Utting, Ian; Cooper, Stephen; Kölling, Michael; Maloney, John and Resnick, Mitchel. "Alice, Greenfoot, and Scratch – A Discussion". ACM Transactions on Computing Education (TOCE) - Volume 10 Issue 4, November 2010, Article No. 17.
12. Flórez-Puga, G., Díaz-Agudo, B., González-Calero,eCo: Managing a Library of Reusable Behaviours. In: Agudo, B. and Watson, I.(eds.) Case-Based Reasoning Research and Development, pp. 92-106. Springer Berlin Heidelberg (2012)

Links

13. Lego Mindstorm, http://mindstorms.lego.com, 2013
14. Unity. Unity: Game development tool, http://unity3d.com, 2013
15. Hivesystem: https://launchpad.net/hivesystem, 2013
16. Yo Frankie!: www.yofrankie.org/, 2013
17. Construct 2: https://www.scirra.com/, 2013
18. Proof of Concept http://code.google.com/p/sven-node-engine/, 2013
19. Blender http://www.blender.org/, 2013