Pipeline-Based Game Development for Blender

Jordi Rovira i Bonet

September 28, 2005

Abstract

This document describes a visual approach to engineering game architectures, in the form of a particular kind of data flow diagrams herein referred to as *pipelines*. This is a work-in-progress report of the implementation of such approach inside Blender. The paradigm is not fully developed yet, so this document will expose the current status and mention the strong points as well as the current drawbacks, and possible alternatives to the identified problems.

Contents

1 Introduction

1.1 Divagations on game development

Computer games are one the most difficult types of software project. The requirements of high performance combined with the crossing of disciplines like rendering, animation, networking, physics, sound processing and artificial intelligence makes the development of a game a real challenge.

Current commercial computer games cost millions of euros to develop and involve teams of hundreds of people, including programmers and artists. However, the games industry seems to be slowly reaching a crisis that might change its fundament and re-formulate its mechanics. An alternative industry of independent self-published game developers is growing slowly, and attracting attention.

In this dual carriage highway, the traditional game producers work hard to always have the cutting-edge technology, while producing new releases of previously successful games in well established genres. On the other side, the independent developers try to produce new concepts and

gameplay, while the technology is left on a second term. They require tools that allow quick prototyping of ideas. These tools have to provide an affordable, flexible and complete framework to allow them starting their development at a high level, but being able to change things at lower level if they require so.

There are many chances that open-source mature projects like Blender will provide them these tools.

1.2 Motivation behind the approach

The visual approach described in this document inherits directly some concepts from the game engine currently implemented in Blender. The "logic-bricks" system has proved to be powerful and useful, however sometimes it lacks flexibility, and can lead to use python scripting for most parts of the project. The pipeline based approach breaks the restriction of the *Sensor-Controller-Actuator* chain allowing arbitrary combination of *modules* that implement any functionality.

This approach also applies the visual diagrams to other processes of the game beyond the in-game logic. Examples of these other processes are the game data construction and the game rendering pipeline, however the system can be used for any process as long as the required modules are provided.

The game data construction process is different for every game. Some require of long calculations to generate normalmaps, lightmaps or environment maps, to precompute visibility, etc., and some others may only require conversion of meshes from the art assets. There are many common operations in these processes, and some of them are typically performed by hand outside the game engine using modelling and rendering suites. The goal of the pipeline-based approach is to provide a visual way to design an automated data pipeline.

The graphics rendering pipeline is even a better example of a process using a limited set of operations combined totally differently in every game. The visual approach should allow defining rendering passes with different material to generate textures and using them to finally produce the presented image, and post-process it.

2 The pipeline paradigm

2.1 Description

A pipeline is composed by a directed graph where the nodes represent actions (and will be referred to as *modules* in this document) and the edges are channels for the data to circulate in (referred to as *links*). The edges arriving to a node are its inputs, and the outgoing edges are its outputs.

The data flows through the modules on the edges and gets transformed through the process of execution of a pipeline. Each module defines the number of input and output connections, as well as the type of data it expects there. A link can only exist between an inputs of a given data type and an output of the same data or a subclass of it. Each module input and output can have any number of links. A module can *emit* data through one of its output connections, and this data will be sent through all the links connected to it.

A module can also define constant parameters that are set externally, and are constant during the execution. Practically, a module is supposed to encapsulate a standard, reusable process. The pipeline creation should be use only to define the architecture based on these processes, not to implement algorithms at low level.

Execution

The whole pipeline is basically executed in the following way:

```
Queue RunQueue;

for each module M do
    M.Initialize(RunQueue);    // Data can be emitted

while RunQueue.NotEmpty() do
    Module M = RunQueue.GetAndRemove();
    M.Run(RunQueue);           // Data can be emitted
```

Whenever a module is run, it can consume data from its input links. When it *emits* data in its initialisation or during execution, the data is sent across the links and all the receiving modules are added at the end of the run queue. This guarantees that a module will always be executed at some point after receiving data.

Not in all calls to *Run* received by a module it will be able to actually do something, as it may require extra data. Whenever a module is run, it is responsability to consume all the data from the inputs, as there are no guarantees that it will be run again.

The execution of a pipeline end when the run queue is empty.

Pipelines as modules

A whole pipeline can be seen as a module itself, with its inputs and its outputs. This allows to structure complex processes in a hierarchy. In the current implementation this is done through a special type of modules that can get data from the parent and bring it into the pipeline and the inverse operation.

2.2 Examples

The following are simple examples illustrating the use of pipelines to implement various processes. Full understanding of the details of each process is not possible without a specification of what each module does, but the general layout can give an idea on how to use this approach to solve problems. The modules are represented by boxes, with the inputs at the left and the outputs at the right.

Collision Detection

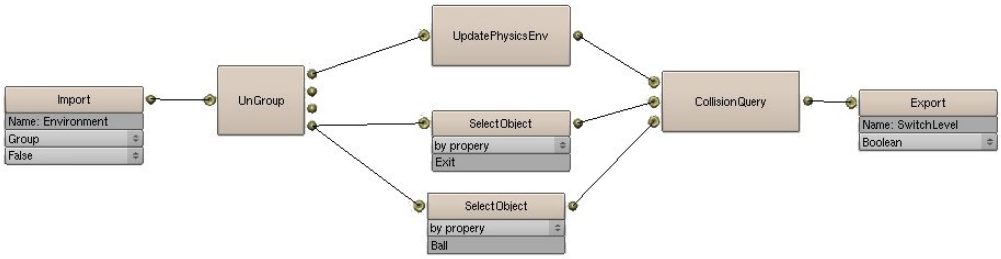

Figure 1: Simple collision detection pipeline.

Figure 1 shows a pipeline that advances the physics simulation and controls the collision of objects and emits a signal to trigger the level change if necessary. Its input comes from outside the pipeline (through an Import module) and consists of a game scene. The game scene is split into

its components in the "Ungroup" module, the top-most output is the physics environment, which is simulated a time delta, and emits a list of object collisions. The bottom link coming out from the "Ungroup" module outputs the list of objects in the scene. Two modules filter the objects with particular properties ("Ball" and "Exit") from the rest, finally, a module checks if any of the objects in one of the group has collided with any of the objects in the other group. A boolean data object is sent as output of the pipeline.

Level selection

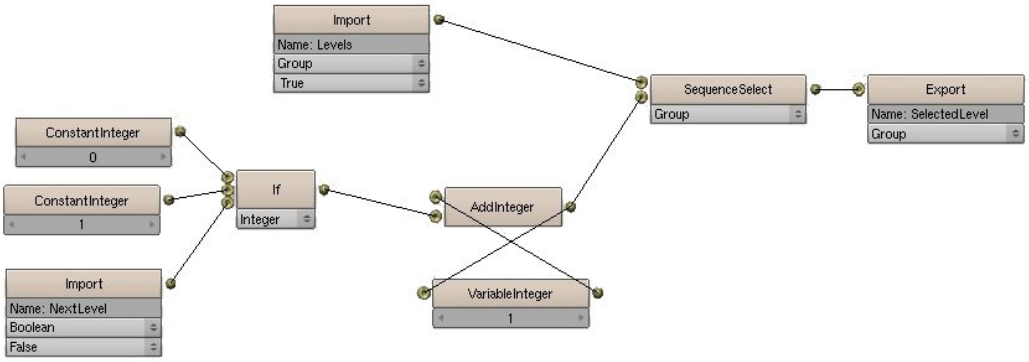

Figure 2: Level selection pipeline.

The next pipeline (figure 2) shows a process that selects a scene (level) from a given set. The main thing to note is the use of a module as a variable, to store a value. The "VariableInteger" module emits the value it is storing at the beginning of the execution, and whenever it receives a new one, it is recorded (but not emitted again yet). The next execution of the pipeline will result on the module emitting the last received value from the previous execution. Apart of this, there is a conditional increment of the last level (based on the boolean signal from the pipelin in the previous example). The bulk of the work is done by the "SelectGroup" module, which selects the n-th scene element from a stream (of scenes).

Platform inclination control

The example in figure 3 shows a complex pipeline that controls the inclination of a platform object, based on keyboard input. This is an example of an overkill solution where probably most of the functionality could be encapsulated in a generic "range control" module.

Rendering pipeline

The next pipeline (in figure 4) represents a rendering pipeline itself. This rendering pipeline uses two passes: one renders all the scene from the light point of view, with a special shader that encodes depth. The second pass reders the same scene from the camera point of view, using the results of the previous pass to perform shadow mapping, and applying normal Phong-lighting shaders.

Data construction pipeline

The pipeline in figure 5 implements all the data conversion process for a game. It takes data directly from Blender and converts it into the engine formats to render and calculate physics. It defines two different paths, one for the "Intro" scene where the objects get converted without

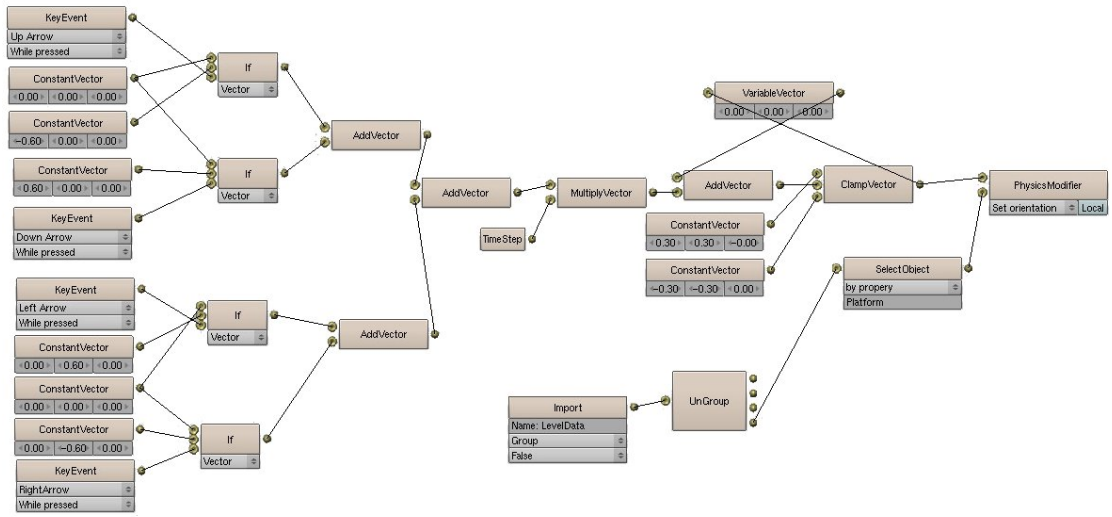

Figure 3: Platform control pipeline.

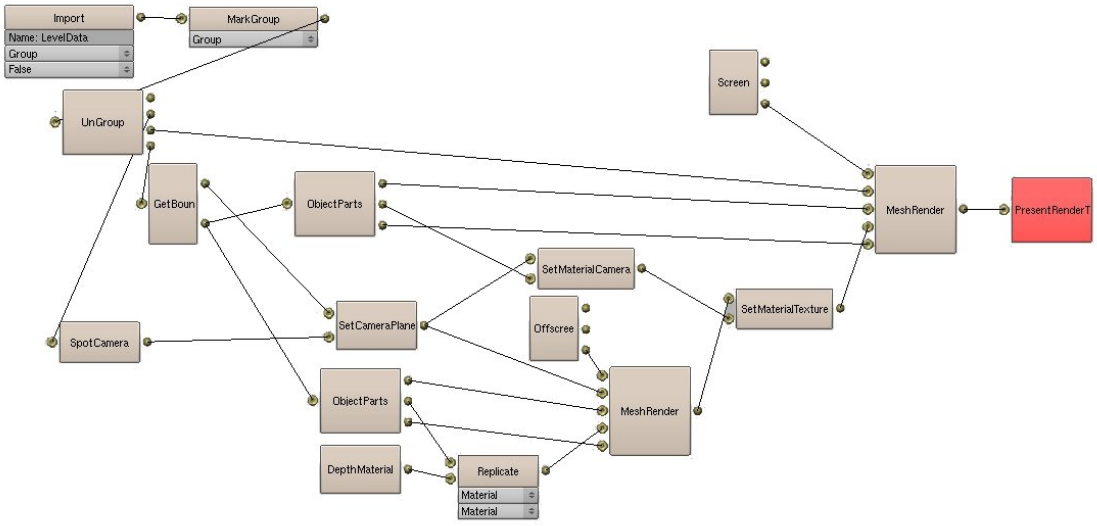

Figure 4: Example of a rendering pipeline.

physics information, and the other for level scenes. This pipelines is quite complex in part due to some of the limits of the paradigm, as exposed later in this document.

Whole pipeline

Finally, all the example pipeline and some more glued together to form the whole system in figure 6.

2.3 Benefits

Software engineering benefits

This kind of approach enforces the encapsulation of processes and a very modular development. Still, all the process have to deal with common data structures, but each process has a well defined input and output and can be reused.

Visual design

The main benefit of the approach is the visual editing of the processes. It is quick and fun to design them by dropping some modules and connecting them.

Parallelism

The pipeline approach opens the door to parallel execution of its processes. Parallelism is feasible between the CPU and the GPU, and also between various CPUs, executing different parts of the pipeline that are not consecutive. This can be of great help for long process of a game data build pipeline like precomputing lighting or normalmaps. Thanks to the graph design is quite intuitive to decide what process can be run in parallel and this could be achieved transparently.

2.4 Limitations

Flexibility

The pipeline model is generic and flexible. It is too generic to control all the processes in a game. For example, the top-level pipeline presented in figure 6 represents the whole architecture of a game. However, the execution of this pipeline cannot follow the same rules as the other pipelines in the examples. Mainly, you want to execute the data build process only one, and then execute the rest of the pipeline every frame. Moreover, if you make a standalone game, you probably want to save the data generated in the "DataBuild" pipeline and load it on game startup.

For this reason, the paradigm has to be broken and special extra rules have to be added. It is easy to foresee that other artificial modifications to the paradigm are required for systems like data streaming, etc.

Efficiency

The pipeline execution has an overhead probably larger than other approaches to do the tasks we are defining with it. The extra cost of running modules whenever they receive any data (even if they are not able to do anything with it while waiting for other data), and the queueing of data in inputs and outputs are other overhead factors. While the intent of the paradigm is to control complex CPU or GPU expensive process encapsulated in its modules, it can be abused to deal with low level control, like in the case of figure 3. In some cases, the possibility of designing visually might be worth the efficiency lose but not in all of them.

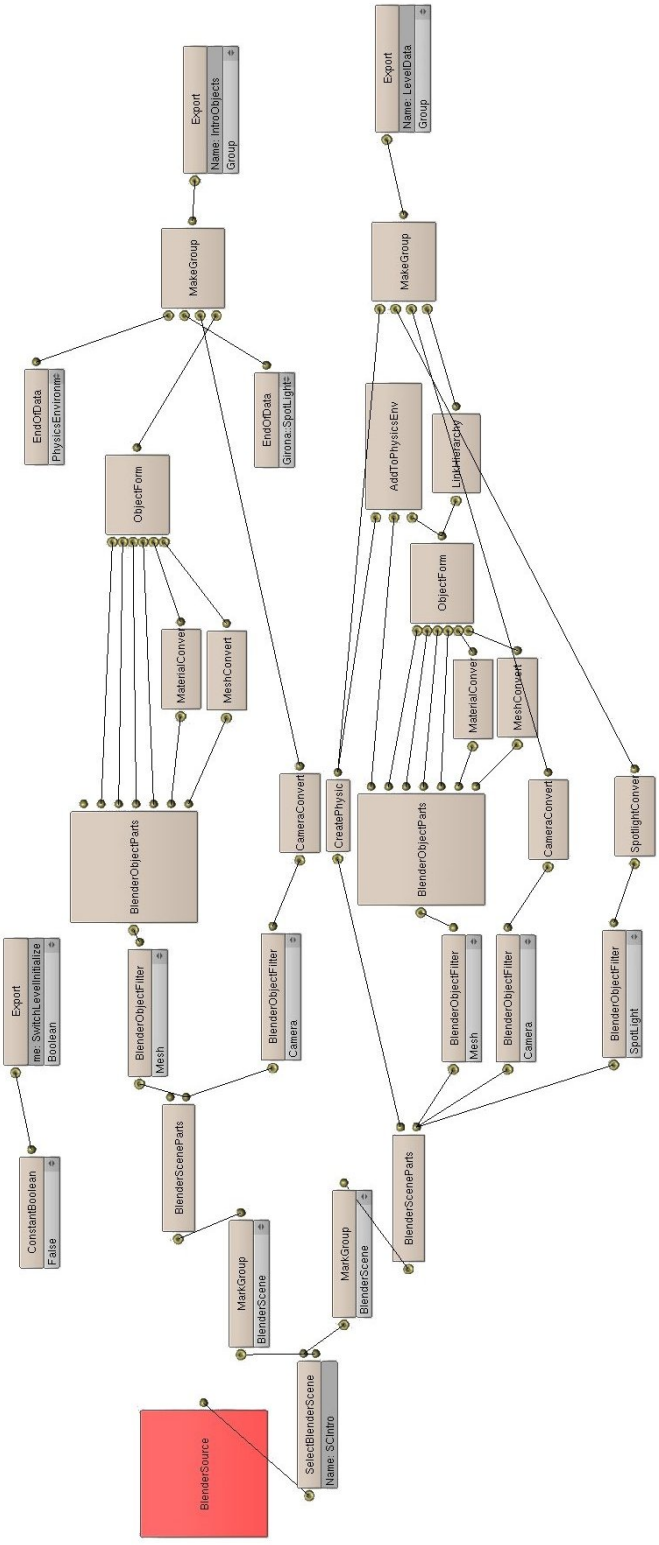

Figure 5: Example of a game data construction pipeline.

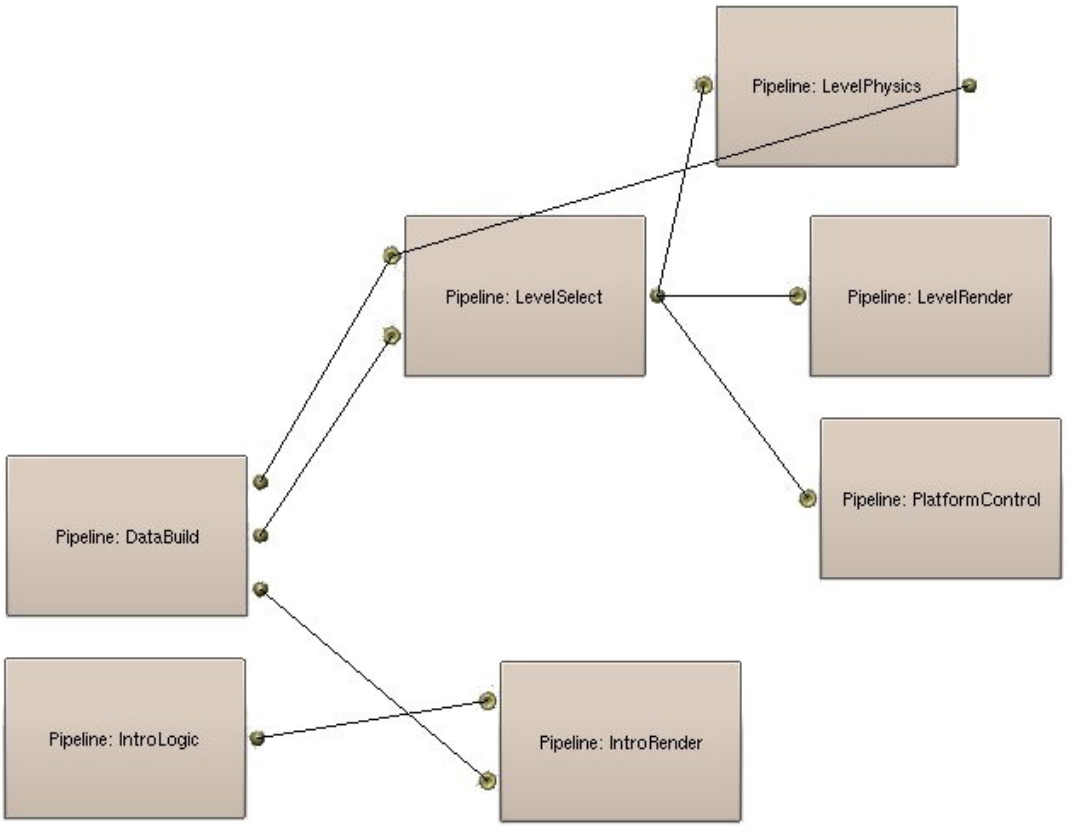

Figure 6: Whole system pipeline.

Paradigm flaws

The major limitation comes from some flaws in the paradigm: sometimes there are many implicitly defined non-intuitive behaviors of the pipelines. For example, in some cases you want your designed pipeline to behave as a digital electric circuit with signals on each link (giving only one value to each input), and in some others you want to have data flowing through them. Maybe the paradigm can be extended in the future to better define the behavior of each link, but right now it is easy to construct visually pipelines that don't behave like they seem at first sight.

Order of execution is not controllable in all cases a side effect of this is that when more than one link arrive to an input of a module, the data order is undefined and potentially could be interleaved, which might be undesireable in some cases.

Another problem of the flow based paradigm is data starving: Data has to flow through all the links every time a pipeline is executed, in order to make the modules run. For example, the boolean value emitted to switch a level (in figure 1) has to be emitted to false every frame, and to true when necessary. This is counter-intuitive, as one would expect it to be able to behave like a "trigger" that only emits data when it is relevant.

These problems are related to the fact that it is not possible for a module to know when it has received all the relevant data. For example imagine a module that receives a stream of objects with a set of properties in each object. If you emit these properties as an output from this module, in the output flow it is not possible to distinguish between the properties of one object and the properties of the next one. The current approach to solve this consist in introducing artificial modules that "mark" the flows with metadata that is propagated through the pipeline. This can be seen in figure 4, where a module "MarkGroups" does this job to allow the modules "MeshRender" to know when they have received all the meshes, and they can emit the render surface to go on in the pipeline. This artificial marking goes against the intuitiveness we are pursuing with this system.

A potential solution to this last problem could the use of "sized flows", where metainformation (invisible to the user) would always indicate how many elements a stream is going to have, but more experiments are required to check the suitability of this for all current uses.

3 Blender implementation

3.1 Status

Due to the experimental nature of this approach, its implementation has been based on targets. A first milestone (codenamed KNIK) was defined consisting of a very simple but complete game that should able to be designed and run. After reaching this, the description of the next milestone is still being decided. The paradigm is corrected and extended whenever it is necessary to implement the milestone.

A simple pipeline editor has been designed in the button-space of Blender (because some modules require their own interface). The pipelines are not stored using the SDNA system in Blender, but externally, in custom text files.

Around 70 modules have been developed, most of them doing very simple and generic tasks like:

- keyboard input
- basic types handling (booleans, integers, reals and vectors, with some arithmetic operations)
- data conversion from blender structures to simpler in-game structures
- some physics capabilities using *Bullet* with sphere and bounding box objects.
- basic rendering capabilities

- framework modules like Import/Export to parent pipelines, or flow marking.

The graphics engine is currently scattered in various modules, but a simple hardware layer will allow to introduce low-level optimizations and to cache the graphics card state, avoiding unnecessary calls and changes.

Also, a very basic standalone player has been developed using GLUT. The pipeline system automatically detects what is the build data and serialises it to disk, along with a modified pipeline including only the run-time logic.

3.2 Other uses inside Blender

One of the main motivations of developing this inside Blender was to be possible to reuse many of the blender processes. There are processes in Blender like the mesh optimization, or the rendering of images that can be part of a game data construction pipeline. It would be quite easy to develop a module with the ability to apply the decimation tool to an object, and emit a simplified version of it. This is particularly easier now that a modifier stack has been coded in Blender: as some encapsulation of the processes has been already done for it.

Other interesting Blender processes useful for a game are its rendereing capabilities (or Yafray's) to compute in-game data, like high quality lightmaps (if render to texture was supported), or environment maps for real-time reflections constrcuted automatically when an object in the Blender scene requires it for the game.

However, the encapsulation in pipeline modules of such features, allows the pipelines to be used inside Blender for non-game purposes, like visually designing macro-operations, or plugins, by visually chaining this modules. Moreover, a user interface for such plugins could be automatically generated by using the unconnected inputs of the pipeline that defines it.

4 Conclusions

A particularly experimental approach to game architecture has been presented. This pipeline based-design doesn't implement anything itself, but is a way to visually structure smaller processes to fulfill bigger tasks. Its enforced modularity makes it very easy to extend and modify single processes without caring about the rest. This is specially important for open source projects based on on-line communities, where developers join and leave project fast, and ocasional contributions happen often. While the efficiency of the framework is not high, it is potentially suitable for any small and medium-sized game project, if used right.

Some benefits and some problems of this approach are exposed in this document with the intention of serving as a base to discuss and do future improvements and experimentation.