Idyll Documentation

Release 0.1.0

Marcus Ottosson

December 27, 2016

Contents

Welcome to Idyll v0.1.0.

Idyll is the description of the ideal setup and situation for integration with Pipi.

Pipi is a decentralised pipeline and as such differ from what you may expect from a traditional method. It is the goal of this guide to enlighten you of this difference as well as introduce you to some of the benefits and reasoning behind why this route was chosen.

CHAPTER 1

Getting Started

1.1 Pre-conditions

This guide will walk you through the ideal setup for use with Pipi.

The goal is to minimise/remove as many pre-conditions (PCs) as possible, but for the time being anything not conforming to the following will require bespoke integration.

- PC1: You should be running either Windows 7 or 8.1, 64-bit with at least 2GB of disk space available.
- PC2: Production should be centered around Maya and Nuke.
- PC3: For production tracking you should be using Asana.
- PC4: Your crew should consist of around 1-10 members with >= 1 technical artist.
- PC5: Production should be full CG and centered around Character-Animation.

Next: Software Configuration

1.2 Software Configuration

Pipi is context-sensitive and treats each running software as a context. This means that any action you take within Software X will be relative to Software X. This way, you and Pipi share knowledge on your current situation so as to save you being overly specific when performing common actions such as loading, saving and publishing your work.

For Pipi to be context sensitive there is the notion of a launcher. A launcher is a minimal application that runs other applications. The only difference between running software via a launcher an running it by hand is the context being set and injected into the application upon launch. By injecting a context, Software X can be made aware of your circumstance and in so doing expose this circumstance to tools used within Software X.

Context in this context (no pun intended) means the modification of environment variables of your system. Prior to running any application - also called *process* - a copy of your environment is made. The launcher modifies your environment prior to running a new process so as to allow each process a copy of a custom environment as opposed to the unmodified environment it would otherwise get from running it by hand.

Some examples of variables modified by a launcher are PATH and PYTHONPATH for context-dependent executables and Python scripts respectively.

Pipi is file-based. This means that all of its configuration is located on disk as plain files and that it will be able to determine your context from looking at where you are within a hierarchy of content.

```
projects
  spiderman
    assets
      Peter
```

For example, this path - /projects/spiderman/assets/Peter - points to the hero asset *Peter* within a project called *spiderman*. To Pipi, this is a context and working within it means to perform every action - load, save, publish - relative to *Peter*.

1.3 Private and Public

Files in a projects are known to Pipi as either Private or Public. Private files are those owned by users and are not intended for use by other than the original author. Examples include scene-files, notes, reference images, tasks. On the other hand there are Public files which are created by one or more users and intended for one or more other users. Files are made public when a user *publishes* his work onto a central location.

```
/Peter/private/marcus/softwarex
/Peter/public/v001
```

This separation is due to quality-assurance and creative freedom.

1.3.1 Filtering

Upon publishing, files may be processed in order to conform to an overarching policy within a central location. For example, the policy at your studio may be for geometry to be built in-line with real world scale. As such, whenever an artists attempts to publish his model at 0.3 nanometer, or 300 million kilometers in diameter, a *filter* may trigger a warning or error depending on how severe the fault is considered and require the artist to re-factor his work in order to qualify for given policy.

This way, supervisors can maintain a level of quality across work produced by multiple artists.

1.3.2 Workspaces

Workspaces are context-dependent directories in which artists save their work. Each artist produces their own unique folder within an additional folder per tool.

```
Peter
  private
    marcus
      softwarex
        myfiles.exe
```

As mentioned previously, Pipi is file-based and context-sensitive. Here, both users and software provide context in addition to the particular asset being worked upon which allows tools to be constructed with awareness of all three.

1.4 Asset Management

One of the major design decisions made with Pipi is the encapsulation of content and meta-content within single branches of directories.

For example, the directory */assets/Peter* contains all data relevant to this asset. This may be what you would expect, however an alternative, perhaps more traditional method is to separate content *produced* by artists, content *published* by artists and the *metadata* associated with a particular assets into three separate branches.

```
/work
  Peter
    myscene.mb
/published
  Peter
    v001.mb
```

Where metadata is stored within a database such as mySQL or mongoDB and only accessible via scripts. One of the reasons for this separation is technical; directories are simply incapable of storing additional metadata and files are limited in how they allow you to modify them. With the advent of Open Metadata however, this limitation is not longer the case and we are again free to join metadata with its content.

As such, in Pipi, the hierarchy looks like this:

```
/Peter
  .meta
    Asset.class
  private
    marcus
      maya
        myscene.mb
  public
    v001
```

This way, you are free to rearrange your content, either permanently or dynamically at any point in time as well as transmit or archive content and always rest assured that no content is ever out of sync.

1.4.1 Tagging

Traditionally, identifying content within a hierarchy is performed via associating an absolute path to keywords or collection of keywords.

```
/projects/spiderman/assets/Peter
```

This path refers to an asset within the project Spiderman, identifying this asset may look like this:

```
assets = {'Peter': '/projects/spiderman/assets/Peter'}
```

However, hard-coding absolute paths may make it difficult to change your mind so further convention may be built:

```
assets = {'Peter': '$ROOT/$PROJECT/$ASSETS/Peter'}
```

Here, keywords have been inserted in-place of actual path-names that may be resolved at run-time. As you can see, there is no longer any mention of the project's name. This way, you are free to re-use tools built upon them in other projects.

We've chosen a different approach. As part of the decentralised philosophy surrounding Pipi, the meaning of content is stored together with the content itself to form content that is so-called *self-describing*:

```
/spiderman
  assets
    Peter
    Mary
    Goblin
  shots
```

```
    1000
    2000
    3000
```

In this example, spiderman consists of 3 assets and 3 shots. Tagging is utilised to place additional meaning into each directory.

```
/spiderman  <-- Project
  assets
    Peter   <-- Asset
    Mary    <-- Asset
    Goblin  <-- Asset
  shots
    1000    <-- Shot
    2000    <-- Shot
    3000    <-- Shot
```

Tagging is performed via a library called cQuery[1]_

1.4.2 Namespaces

For context sensitivity, Pipi treats the current working directory as namespace. This means that while your are located within a certain directory, the actions you take within this directory will be relative this directory.

This also has an effect on the layout of your directories.

```
/projects/spiderman/models/Peter
/projects/spiderman/rigs/Peter
/projects/spiderman/shaders/Peter
```

In the above example, models are collected within a common directory and each asset separates their corresponding model by name. In this scenario, metadata stored at the *models* directory of *Peter* will not be visible with metadata stored with *rigs* and as such will either need to be duplicated or remembered.

Considering that *Peter* is more likely to pertain metadata than *models*, a more efficient layout may look like the following:

```
/projects/spiderman/Peter/models
/projects/spiderman/Peter/rigs
/projects/spiderman/Peter/shaders
```

In this example, we may associate metadata with *Peter* and it would remain consistent whether you are exploring his models, rigs or shaders.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search