The Pipeline

Goals and Non Goals

Goals

- Create tools and processes to support the production teams for the BUGJam participants.
- Iterate on the tools to improve them for the next BUGJam and beyond.
- Teach developers, artists, production managers, etc about studio pipelines and what they can do.
- Produce open source code that can be used in other projects.

Non-Goals

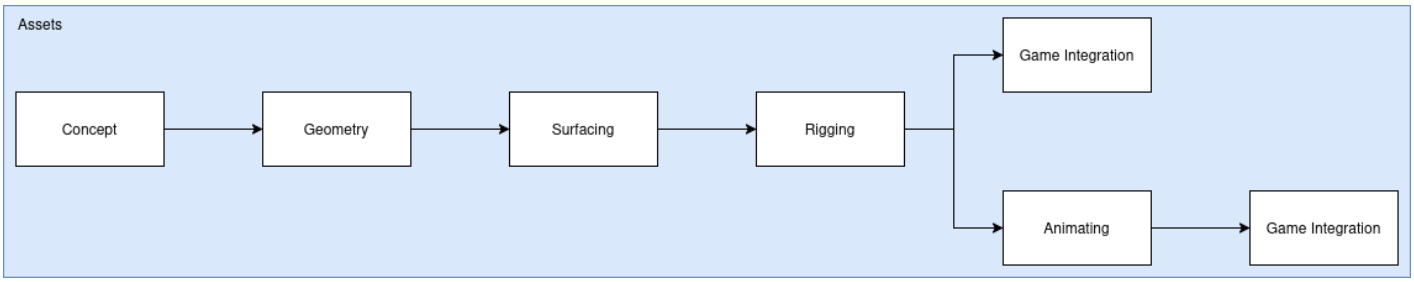
- Support non-open source tools such as Maya or Photoshop. The pipeline **should** be able to support them and not be completely designed around Blender and Godot, but no effort will be made to support applications outside of the BUGJam list of supported applications.
- Support project sizes outside of the BUGJam projects. The pipeline **should** be built in a way to allow it to scale to larger projects, but no effort will be made to support projects beyond the scope of the BUGJams.
- Monetize the pipeline. If that were to happen, then there would need to be a discussion with all the contributors.
- Have a fully functional pipeline right away. This is all done by volunteers and the pipeline will grow as production lessons are learned and people contribute.

Design

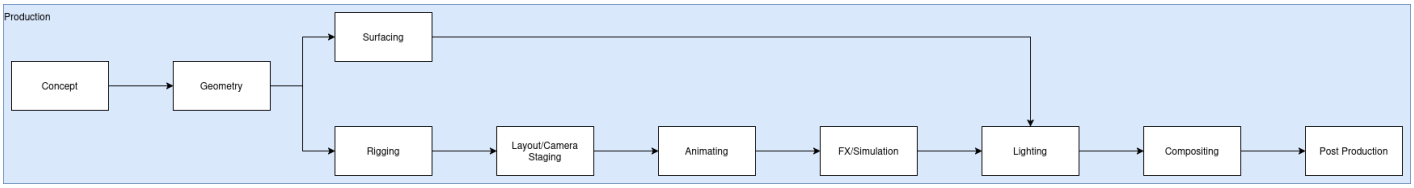
The goal of the pipeline is to support the production team working on the BUGJam. While we will not be able to have a full pipeline from the start, we can at least start off with a small pipeline and grow it and this document. All contributions to the proposed long term designs are highly welcome.

Production Flows

Games



Film/Episodic



Infrastructure

Each phase represents a different BUGJam project.

Phase 1: Minimal Game Pipeline

- Deploy: We need to be able to deploy the pipeline to each jam participant.
  - Possible solutions

        - Python wheels and upload to the Forgejo package repository, then creating a virtual environment for each of the applications and installing the packages from the Forgejo repository and PyPI.
        - Rez
- Bootstrap: We need to be able to launch the applications in our environment with the proper environment variables, application flags, resources, etc.
    - Possible solutions
        - Shell scripts
        - Launcher application
- Asset Workspace: We need to be able to create a location for an asset to save to.
    - Possible solutions
        - A small utility in Blender to create asset directories based on a given name
- Publish: We need to be able to publish and sync work from production members.
    - Possible solutions
        - Plain Git commands

Phase 2

- Entities tracking: We need to be able to track entities and start to know more about what everything is/what needs to be done.
    - Possible solutions
        - Kitsu
        - Roll our own entities database
- Context management: We should keep track of the current context someone is in when using their DCC so we can easily manage the environment (for example, have the asset library default to show only assets that are part of the shot).
    - Possible solutions
        - Roll our own context manager
- Publish: The publish tool will need to become more production friendly and handle tracking
    - Possible solutions
        - Pyblish
        - Roll our own
- Validations: We should have validations happen at every step of the pipeline.
    - Possible solutions
        - Pyblish
        - Scott's checks framework
        - Roll our own
- Workspace: We should support creating loading scenes via a workspace loader
    - Possible solutions
        - Roll our own

Phase 3

- Metadata: We should start tracking more information about assets, shots, etc. This may include sidecar files for all of the publishes
  - Possible solutions
    - Universal Scene Description
    - Roll our own (sqlite, json, etc)
- Metrics/Error tracking: While we need to make sure we're extremely careful about collecting potentially personal identifying information, being able to track errors and bottlenecks is extremely useful. This would need to be a big discussion with everyone about who gets access to the data (assuming that people are comfortable with the information being stored), and how to do this in a way that'll make people feel comfortable (show them what information would be transmitted, opt into data collection, etc).
- Shot/Map assembler: We should have some automation that could create a new shot/map from scratch in our DCCs using assets that are assigned to the shot.
- Asset library: While Blender does have an asset library, we can add some logic that connects to the asset database, and handle different cases (loading assets to work on or work with).

Game folder structure

This is the guide on how to name things, where to put them, and over-all how we intend to organize the project files.

Considerations

The game project's file structure should achieve several key things:

- Organize multiple repositories into folder structures that can co-exist.
  - One directory for the Godot project, a separate directory for the art source files that are created outside of the engine.
  - These directories will each be a Git repository and should conform to the hyphenated naming convention.
  - These names should make sense in and out of context, the top-level folder name should make sense when viewed as a tab in a Git GUI e.g:
    - **example-game**
    - **example-art**
    - **example-audio**
  - Splitting the repositories avoids having to sync large amounts of irrelevant content for your role on the project:
    - Programmers / Dev Ops / Build Bots should only need to sync the **example-game** repository.
    - Artists / Animators can sync **example-art** along side the **example-game** repository.
    - Musicians / Foley Artists can sync **example-audio** along side the **example-game** repository.
  - The folder structures should work without using Git sub-modules, since sub-modules add a lot of complexity and confusion without really adding value.
- The folder structures in these directories should mimic each other 1:1
- Tools and pipelines should be able to export / import content by simply swapping the root, e.g:
  - **example-art**/art_assets/characters/main_character/main_character_diffuse.png
  - |--->**example-game** /art_assets/characters/main_character/main_character_diffuse.png
- Ship-able game content should be the only thing going into the Godot project resource folder in the game repo.
- Art source content from tools like Blender, Gimp, Krita, etc... should be kept in the art repo.
- Music production files for a DAW should be kept in the audio repo.

Naming Conventions

- The repositories should use **lowercase-hyphenated-case**
- Files and folders should use **snake_case** unless otherwise required by third-party tools or conventions for the specific file e.g. README.md
- No spaces in file names.
- All directories should be plural:
  - "characters" not "character"
  - "environments" not "environment"

Compromises

No folder structure is perfect, but this structure should lead to the fewest problems. Here is a list of known compromises we've decided to accept:

Top level folder names repeat the game's name even though they are already inside of a top-level folder named after the game:

- **example**
  - **example-game**
  - **example-art**
  - **example-audio**

While this introduces redundancy, folders don't cost anything, so this is okay.

Some intermediate sub-folders are empty and seem to be useless, but they keep the hierarchies between directories in perfect sync. For instance, **example-art** may never end up with content directly inside of it, most if not all content will be inside of its first sub-folder **art_assets** i.e. example-art/art_assets/ This is on purpose, since it ensures the game content is at the same hierarchical indentation level to the source content, making pipelines and paths extremely easy to work with; just swap the root and you're done!

You ought to have a directory on your computer for "**BUGJam**" projects. You can place this anywhere on your computer with a few considerations:

- Do not sync this directory with OneDrive, iCloud, Google Drive, or any other backup tool, this will break the Git repositories inside the folder since these services interfere with Git.
- Don't nest the folder too deep so you avoid file name length issues:
  - Good (Windows style path): `C:\Users\xgreer\Projects\BUGJam`
  - Good (MacOS / Linux): `/home/xgreer/Projects/BUGJam`
  - Bad: `/home/xgreer/Projects/Extra Projects/Seattle Blender User Group/Game Projects/BUGJam`

Within the BUGJam directory there will be folders for each BUGJam project. For a project called '**Genesis**' we will have a project folder called 'genesis' that serves as the root for all content necessary to develop the game.

File Tree Structure Example

```
BUGJam <!-- Top-level folder for the entire studio. -->
├── genesis <!-- Top-level folder that contains all repositories for the game. -->
│ ├── genesis-art <!-- Repository for large art content, stored outside of the Godot project.
-->
│ │ ├── .forgejo <!-- Hidden content used for resources on Forgejo -->
│ │ │ └── banner
│ │ │   └── genesis_art_banner.png
│ │ ├── art_assets <!-- Source files for creating art -->
│ │ │ └── characters
│ │ │   ├── main_character
│ │ │   │ ├── animations
│ │ │   │ │ └── idle.blend
│ │ │   │ ├── main_character_diffuse.png
│ │ │   │ └── main_character_rig.blend
│ │ │   └── monster
│ │ │     ├── animations
│ │ │     │ └── walk.blend
│ │ │     ├── monster_diffuse.png
│ │ │     └── monster_rig.blend
│ │ ├── audio_assets <!-- Source files for music and sound effects -->
│ │ │ ├── dialogue
│ │ │ ├── music
│ │ │ └── sound_effects
│ │ └── pipeline <!-- Pipeline tools for producing content outside of the game -->
│ │   ├── addons
│ │   ├── extensions
│ │   └── scripts
│ └── genesis-game <!-- This is the Godot root for resources: 'res://' -->
│   ├── .forgejo <!-- Hidden content used for resources on Forgejo -->
│   │ └── banner
│   │   └── genesis_game_banner.png
│   ├── addons <!-- Godot plugins -->
│   ├── art_assets <!-- A 1:1 mirrored file structure with the genesis-art repository. -->
```

```
|   | ├── characters
|   | | ├── main_character
|   | | | ├── main_character_mat.tres <!-- Material -->
|   | | | ├── main_character_diffuse.png <!-- Texture -->
|   | | | ├── main_character_diffuse.png.import <!-- Import settings and UID -->
|   | | | ├── main_character_rig.bin <!-- Binary data for the model file -->
|   | | | ├── main_character_rig.gltf <!-- Model file -->
|   | | | └── main_character_rig.gltf.import <!-- Import settings and UID -->
|   | | └── monster
|   | |   ├── monster_mat.tres <!-- Material -->
|   | |   ├── monster_diffuse.png <!-- Texture -->
|   | |   ├── monster_rig.bin <!-- Binary data for the model file -->
|   | |   ├── monster_rig.gltf <!-- Model file -->
|   | |   └── monster_rig.gltf.import <!-- Import settings and UID -->
|   | ├── common
|   | | └── textures
|   | |   └── blue_noise_64.png
|   | ├── environments
|   | | ├── rock
|   | | | └── [...]
|   | | └── wall
|   | |   └── [...]
|   | └── interactables <!-- Core puzzle elements for designing levels. -->
|   |   ├── coin
|   |   | └── [...]
|   |   └── door
|   |     └── [...]
|   ├── audio_assets <!-- A 1:1 mirrored file structure with the genesis-audio repository. -->
|   | ├── dialogue
|   | ├── music
|   | └── sound_effects
|   ├── entities <!-- Complete assets that are ready to drag and drop into a level. -->
|   | ├── characters
|   | | ├── main_character.tscn
|   | | └── monster.tscn
|   | ├── environments <!-- Stuff without direct gameplay mechanics. -->
|   | | ├── rock.tscn
|   | | └── wall.tscn
|   | └── puzzle_elements <!-- Core puzzle elements for designing levels. -->
|   |   ├── button.tscn
```

```
│   │   └── pusher.tscn
│   ├── levels <!-- Playable levels. -->
│   │ ├── level_packs <!-- Data assets that organize levels into packs. -->
│   │ │ ├── world1.gd
│   │ │ └── world2.gd
│   │ ├── epilogue.tscn
│   │ ├── intro.tscn
│   │ └── open_world.tscn
│   ├── scripts
│   │ └── player.gd
│   ├── shaders
│   │ └── iris.gdshader
│   ├── ui
│   │ └── icons.png
│   ├── .gitignore
│   ├── icon.png
│   ├── project.godot <!-- Main Godot project file -->
│   └── README.md
└── logo <!-- Example of other stuff for the studio that is not directly related to the game.
-->
```

Godot Tools

WIP Page on tools in Godot

In the top right are a few non-standard elements:

Left-most chooses which level gets launched with F5 (which technically is GAME_ENTRY but then immediately loads something else). By editing `ignore/debug_config.tres`, you can scenes. The advantages of this are that it's per-user, very convenient to change levels, only runs when launching from the editor so you can't accidentally set the wrong level and makes it much more convenient than F6 because you might be also working on a prop/player/whatever and it's really annoying having to click back to the level for each tweak.

TBD Spawn selector / toggles

Code Organization and Rules

This document is currently an unorganized scratch pad of thoughts on what the future pipeline should look like. Discussions are very welcome.

The intention of the core pipeline design is to lower developer friction when developing, easy to deploy, and support arbitrary front ends (Blender, Godot, Krita, small CLIs, etc).

Rules

The rules should be followed unless it makes more sense to not follow them. They are there to make it easier to reason about the code and where code should live.

- API first: All functionality should be independent of UI, and the UI code should only handle displaying and manipulating the data that gets passed to/from the APIs. This is to allow for greater reuse of functionality across the pipeline. Blender's operators are considered UIs in this context. Functionality that have to be in the lower API includes business logic such as communicating with a database, creating meshes, modifying a scene, etc. Functionality that can live in the UI include transforming the data into a human readable format (unless it is a very generic task that will likely be shared such as converting a number into a distance, size, temperature, etc with units), extracting information from a Blender context to pass into the API, etc.
- Invalid states should be impossible: If a tool expects a certain state, and that state is invalid, then that tool must not allow the user to progress until the state is valid. For example, if a publish requires a valid object or collection to be selected, and for comments, then the publish button must be disabled until the conditions are valid. The API should also not assume the data is valid and validate/throw an error. We could use the new type pattern (See https://doc.rust-lang.org/rust-by-example/generics/new_types.html for an example) to do the validations up front and encode them in a class that must remain valid at all times.
- Code should live in their appropriate place: Generic code that does not interact with applications should live in the core, where application specific code should live in the application areas. For example, the generic publish code should live in the core pipeline, where application specific publishing such as rigs, animation, geometry, etc should live in the application area. Also, the libraries are split between the absolute core such as path resolution, and more specific such as turntable generation and management.
- Static where possible, dynamic when necessary: Whenever possible, the code should be static. This could mean functionality, environment resolution, static types, etc. Static code is easier to manage. Code should only be dynamic when static could not solve the issue

better than dynamic.
- Simple: We should strive for simplicity both for lower maintenance work and easier reuse.

Project Structure

```
/pipeline/
  blender/
    src/
      operators/
      ui/
      pipeblend/
        core/
        {tool_lib}/
      __init__.py
  core/
    src/
      pipecore/
        core/
        {tool_lib}/
      __init__.py
```