**Vessoul**

**Team Spinning Chairs**

**GAM200**

**Fall 2014**

**Esteban Maldonado (RTIS)** - Producer, Graphics Programmer

**Judy Cheng** **(RTIS)** - Physics Programmer, Audio Programmer

**Steven Galwas** **(BSGD)** - Co. Game Designer, Gameplay Programmer

**Henry Morgan** **(RTIS)** - Tech Director

**Ian Aemmer** **(BAGD)** - Lead Game Designer

**Nathan Jaron Bernard Hansen** **(BFA)** - Lead Artist

**Daniel Maisonet** **(BFA)** - Artist

**Claude Alaras** **(BAMSD)** - Composer, Sound Designer

**Table of Contents**

Overview of the Spin Engine………...…….03

Major Object Components……..….…….06

Graphics Implementation……………….…..09

Behavior Implementation……………….....10

Physics Implementation……………….….….11

Multiplayer Implementation………..……..12

Coding Method…………………….……………..13

Debugging…………………………………………...14

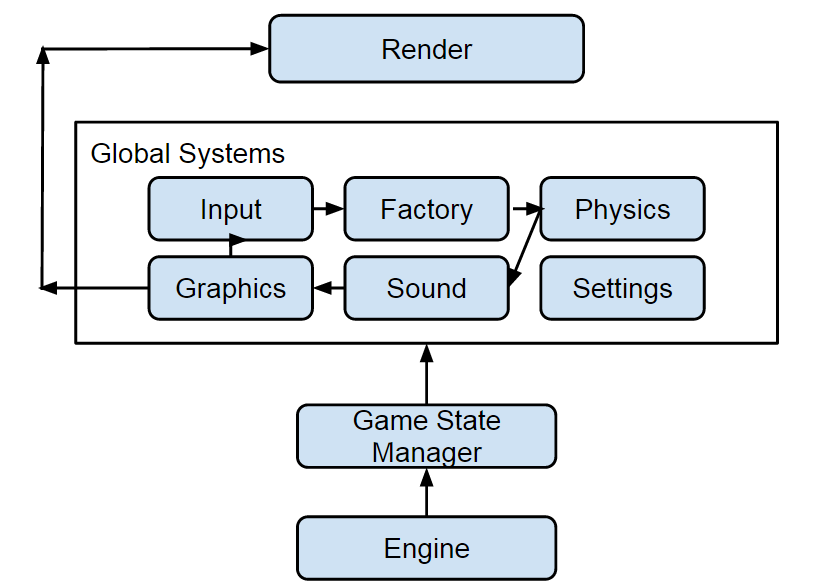
Tools…………………………………...……………...15

Appendix ..……………………..…………………...16

**Overview of the Spin Engine**

**Global Components**

The Spin Engine has six global components (referred to hereafter as Systems): Input, Factory, Physics, Sound, Graphics, and Settings. The first five systems are updated in a loop each frame in the order they are listed. All five are instantiated and updated in the Engine class, which contains a game state manager and all code for system startup and shutdown. The Settings class loads all tweakable variables from a file (settings.json) and is accessible by any other class requiring one of the variables.



**Input:**

The Input system contains a global messaging system; any object may receive communication from it by obtaining an InputListener component. Messages are sent to all registered Listeners. It is up to the InputListener’s sibling components to check for the relevance of any messages received.

The input system checks for changes in the keyboard and gamepad states and sends a message to all InputListeners with that information. The system tracks whether a key/button is Down (has just been pressed this frame), Pressed (has remained pressed from a previous frame), or Released (was just released this frame). The system checks periodically to see if controllers have been added, and only checks for updates from controllers that have been connected.

The Input system contains playback functionality. If recording is enabled, all controller (not keyboard) input will automatically be saved to a file (with the convention “LevelName-HourMinute.json”, I.E. “Level\_2-0215.json”) after any given level begins. Recording will stop on completion of that level, and resume with a new file upon beginning (or restarting) a level. If playback is enabled and the appropriate file exists, all controller input will be ignored and the system will use the recording file to determine inputs. Playback and Recording can be enabled simultaneously, but all this will do is circuitously copy whatever recording is being played back.

**Factory:**

The Factory is contained within the FactoryManager class. It creates, manages, and destroys game objects. It interacts with the Graphics system to register and deregister Sprite Renderers, and interacts with the Physics system to register and deregister Rigidbodies. FactoryManager has four sub-classes: JSONLoader, JSONSaver, ObjectData, and LevelData.

The JSONLoader and JSONSaver classes encompass the Spin Engine's serializer- it is capable of saving to and loading from JSON-formatted files. FactoryManager uses JSONLoader to load a master .json file (Assets/master.json) containing level names and asset locations, and from there can be prompted to load any level by name. All file I/O is accomplished through the factory's serializer.

The ObjectData class loads all asset-dependent data (currently textures and Sprite data), and then creates Sprite components when prompted. The LevelData class loads all level names and level file locations, and can be prompted to load and provide all information required to construct a level upon receiving a level name.

The FactoryManager class is responsible for loading and unloading levels. After prompting LevelData for level data, it uses the data to construct a tilemap (a binary array, and simple game objects containing only a SpriteRenderer) and all custom game objects. All game objects share a single vector in FactoryManager, which is cleared when a level is unloaded.

Every system can access the Factory through a global accessor class. This is used in various components to access tilemap info, create and destroy game objects at run time, and (in the case of the Game State Manager) to change levels.

**Physics:**

The Physics system is contained within the PhysicsManager class. It contains two vectors: one of Colliders, and one of RigidBodies. When a Collider and a Rigidbody are attached to an object, it is automatically added to the system. Colliders are only checked when set to active; otherwise they will be skipped over. If two objects have static rigidbodies (are immobile), that collision is also skipped. Every contact is put into a list. Resolution will only occur if and when contacts are made.

Only the Factory can access Physics, and only does so to register and deregister Colliders and Rigidbodies.

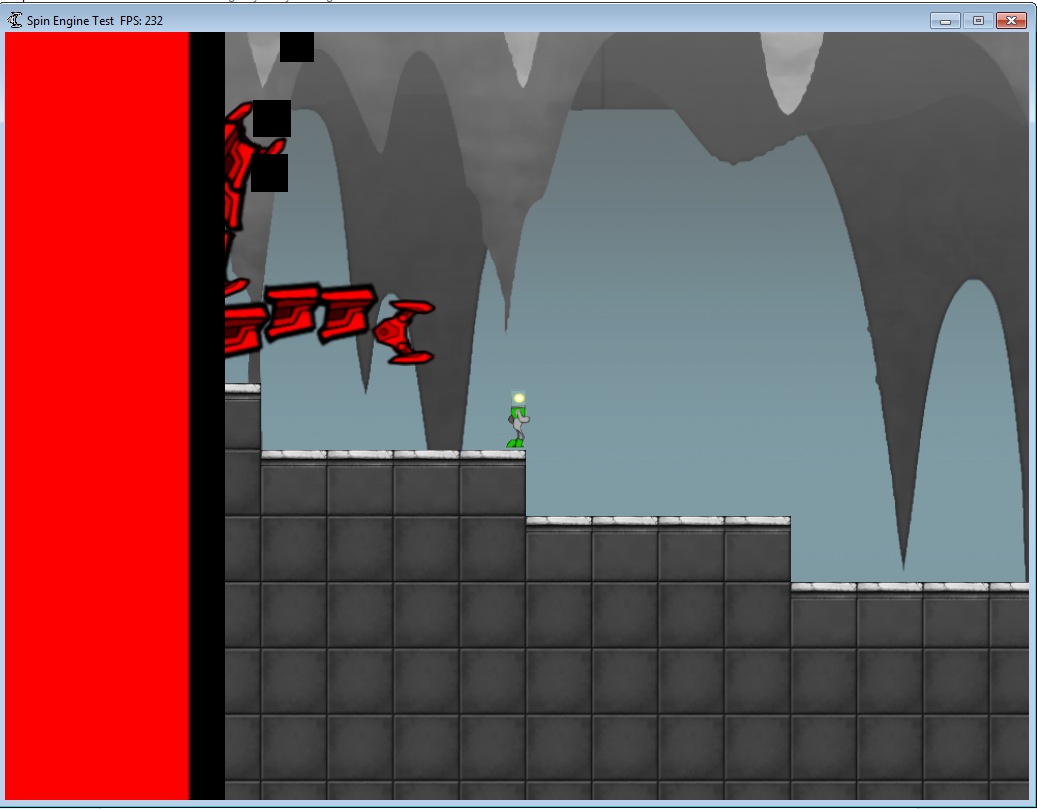
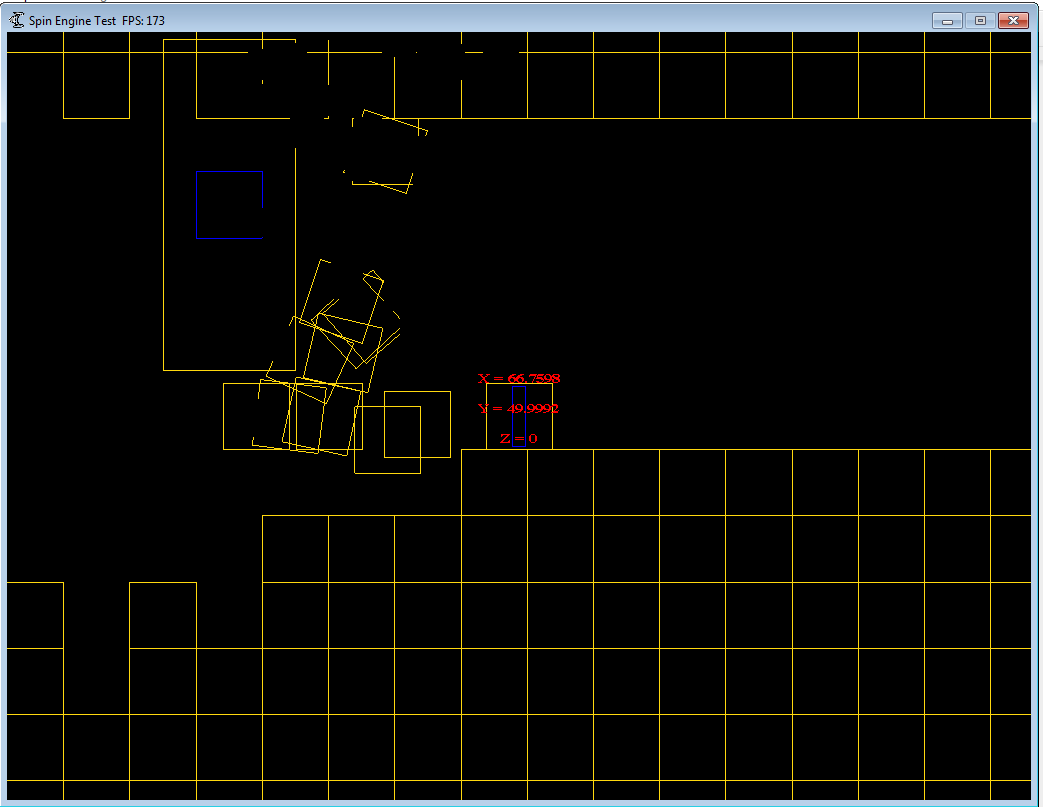
**Sound:**

The Sound system is contained within the SoundManager class. SoundManager contains all integrated FMOD functionality, and is accessible through the SoundEmitter component. SoundEmitter provides playback functionality to sibling components; siblings may play, stop, pause, and change the volume of any event they have been given the name for.

**Graphics:**

The Graphics system is contained within the GraphicsManager class. It contains three vectors of pointers to Sprite Renderers. GraphicsManager goes through each of these vectors in order and draws each sprite renderer. Multiple vectors are used to allow finer control of z-layers; the three vectors used are for Background sprites, regular sprites, and UI sprites. (The UI layer is also used for any other sprites that need to appear above the gameplay plane.)

GraphicsManager contains a Debug Drawing mode which draws the wireframe models of 2D sprites as well as the collision boxes of all objects that have them. The Debug Drawing mode also displays the x-, y-, and z- coordinates of the player.

Regular Rendering Debug Drawing

FactoryManager is the only system to interact with GraphicsManager, and only interacts to register SpriteRenderers. Each Sprite Renderer interacts with GraphicsManager by calling to have its pointer removed from the system on its destruction.

**Settings:**

Settings uses a temporary instance of the JSONLoader class to load in variables from the master settings file (Assets/settings.json). Any object or component can then access Settings to retrieve the value of a given variable. Variables are automatically added to the TweakBar (see Tools), but can be set to not be visible once all desired tweaks are complete.

**Major Object Components**

Non-global components are made part of a composition within the Factory. All components are provided with a pointer to their parent object when they are created. Each component can use this pointer to get any other component from the parent. Components are updated when their parent object is updated by the Factory, but in cases where components are accessed by other systems (as is the case with Sprite Renderers, SoundEmitters, and Rigidbodies), the most significant functionality (such as the Draw function in SpriteRenderer) is called by the relevant non-Factory systems.

**Transform:**

The only required component, Transform stores game object position, rotation, and scale. (RigidBody also tracks object position; when both are present, and position needs to be directly modified, RigidBody’s position field should be altered instead of Transform’s, as the Rigidbody will update Transform according to its “position” field.)

**SpriteRenderer:**

This component contains an instance of the Sprite class, which stores the pointer to a sprite's texture as well as the current frame of an animation, frame time, and other graphical data. Drawing is accomplished via the Draw function, which is called by the Graphics Manager.

**Camera:**

Camera contains all functionality for calculating view and projection matrices, and tethers the camera to its parent object. (There should only be one instance of this component.)

**ParticleEmitter:**

ParticleEmitter allows particle creation at the parent object's location.

**ParralaxBG:**

Exclusively within Camera. Maintains multiple parallaxing backgrounds.

**SoundEmitter:**

SoundEmitter allows sibling components to trigger events (sounds), acting as an accessor to SoundManager’s functionality. Siblings may play, stop, pause, and change the volume of any event they have been given the name for.

**RigidBody:**

The RigidBody component contains information on the object’s mass, inverse mass, friction, inertia, velocity, and position. It manages velocity, acceleration, and position for physics-based movement. It forms the basis for collision detection. It can be either dynamic or static; static rigid bodies will not move, but can and will collide with non-static rigid bodies. A RigidBody can be made a ghost by setting the isGhost property to true; ghosts will detect (but not resolve) collisions with other rigidbodies.

PhysicsManager can only work on objects with a RigidBody- objects without one will not be added. RigidBody must be given a Collider in its constructor; this Collider must also be a sibling component.

RigidBody contains trigger functionality that allows its sibling components to register callback functions with it, which will be called whenever any other object collides with the RigidBody’s parent.

**Primitive (AABB and Circle):**

AABB and Circle are the Physics system’s two collider shapes- an Axis-Aligned Bounding Box and a Circle. Both derive from the Primitive class. Only one should be added to any given game object; to this end, only one can be registered with its parent’s RigidBody. (This is done upon construction of the RigidBody.)

**CollisionDelegate:**

The base class inherited by any component wishing to register a callback function. Contains a single virtual function (OnCollision) overridden within the derived class. Registers itself with the parent object's RigidBody on initialization.

**TileMapCollision:**

Adds tilemap-based collision to the parent object. Non-physics based; checks current position against tile position in FactoryManager's tilemap array, then snaps to edge of any colliding tiles.

**TileMapDetection:**

Adds tilemap access to the parent object. Does nothing to prevent collision; instead, provides ability to add or remove tiles (or to remove them from the tilemap, but acquire access to them as game objects).

**Listener:**

Registers itself with the global messaging system to receive messages. Other components can access the listener to check for specific inputs. The input system supports keyboard-specific functionality of KeyDown and KeyPressed.

**Graphics Implementation**

The Spin Engine’s graphics are implemented with DirectX 9. We are using fixed functions. Each texture (Assets/Textures/Tile\_Ceiling.png, etc…) has an equivalent Animation file (Assets/Animations/Tile\_Ceiling.json, etc…) containing all information necessary to create an instance of the Sprite class using that texture. Since each instance of the Sprite class contains unique information about its current animation state, the Factory creates a new instance of Sprite for each game object, but maintains a single loaded instance of each actual texture.

We plan to implement shaders, both to offload work onto the GPU and to allow us to use normal maps for a planned 2D lighting system.

**Behavior Implementation**

The “Wall of Death” is the sole behavioral entity in the game. Its speed is constant, but the “claw” movement and grabbing of tiles is algorithmic. Each claw chooses its own target. Targetable objects have a component (WallTarget) on them that adds them to an array of possible targets. (There is an additional tilemap array specifically for wall-targetable tiles, but this is currently unused.) The wall chooses something from the object array that hasn’t already been targeted. If there are fewer targets than there are hands available, it will instead fire in a random direction, and choose the first tile or game object it hits as its target. If a claw hits a tile, it will grab it and drag it back, keeping it alive as a game object until it hits the wall but removing it immediately from the tilemap arrays. If the claw hits a player, it will drag the player back to the wall (and the player’s death). The wall’s speed is proportional to the player’s distance to it- at greater distances, the wall will move faster.

The actual path that a claw follows is determined by creating a sin wave and then rotating it to pass over the target. As the claw follows this sin wave, it creates wave-aligned “links” behind it. When it returns, the claw and all links go straight back to the origin.

**Physics Implementation**

The physics system utilizes two basic shapes as collider components: an axis-aligned bounding box and a circle. It has two types of detections: Broad collision detection, which is quick and simply returns true or false, and Narrow collision detection, which is more detailed, calculating the penetration depth, collision normal, and number of contacts on a collision. Detection uses the concept of the Separating Axis Theorem when checking if two objects overlap. Forces are integrated using the semi-implicit Euler method. Resolution is a simple impulse system that prevents objects from continuing to intersect each other; objects bounce off each other.

PhysicsManager uses the Manifold class to store information about a collision between two objects. The information within is used to calculate resolution.

Tilemap collision is not handled within PhysicsManager; rather, the TileMapCollision component handles all checks and resolution for a given object against the binary collision map created by the Factory.

**Multiplayer Implementation**

ControllerComponent has support for input from multiple gamepads, which we can use to allow cooperative or competitive play in the future. We have no specific plans for further implementation at this time.

**Coding Methods**

There are no strict file naming conventions, but all source and header files must be sorted in folders (within the Source folder in the main directory) by category:

* Core: the Engine class and core components.
* Factory: The factory, the serializer, and Settings.
* Gameplay: All gameplay components.
* GameState: The game state manager and basic States.
* Graphics: GraphicsManager, Camera, particle-related code, graphics-related code.
* Includes: External library header files.
* Input: InputManager, and all code/components related to input.
* Math: All math code.
* Physics: PhysicsManager, and all code/components related to physics.
* Precompiled: The precompiled header file and the empty precompiled.cpp.
* Sound: SoundManager, and all code/components related to sound.
* Utilities: Miscellaneous cross-category code, such as our MemoryManager class.

The maximum length of a line of code is 110 characters.

We use Mercurial with the TortoiseHG client for source control. There are three team rules regarding its use:

1. At no time may the “default” branch be broken.
2. No branch may go more than a week without being merged back into “default”.
3. At least one commit message on a branch must meaningfully explain its purpose.

Documentation is not standardized, but is required on all code. Brief code reviews are performed each week to ensure legibility between all team members.

**Debugging**

The Spin Engine has debug drawing, gameplay playback, and a tweak bar. All three features can be enabled or disabled at will. Additionally, when not in Release mode, the current frame rate is always displayed at the top of the window.

The debug drawing system can draw the wireframe models of 2D sprites as well as the collision boxes of all objects that have them. When the debug drawing system is activated, we display the x-, y-, and z-coordinates of the player. Debug drawing can be activated by pressing F2, and then F3 for an alternate view.

The playback system records player input for a given level and saves it to a .JSON file named with the level name and time of recording. The time of recording must be removed from the filename to play back the gameplay; when playback is enabled, the playback system searches for a single file with just the level name, loads that, and uses it as the input for that specific level.

The tweak bar (AntTweakBar) is primarily used for tweaking variables, but can be used to watch them at runtime. This provides a viable alternative for viewing the state of variables when pausing the game and stepping through break points in Visual Studio is not desirable.

**Tools**

**Third-Party Tools**

**AntTweakBar:**

AntTweakBar is an overlay that allows editing of variables at runtime. The tweak bar is integrated into the Settings system- values retrieved from Settings are added to the tweak bar unless otherwise specified. One instance of each tweakable variable exists- all classes retrieving the value acquire a pointer to that instance.

**Tiled:**

The third-party editor used to create all level files. It requires an up-to-date sprite sheet, which is currently used exclusively by it (Assets/Textures/Tiled\_Spritesheet.png). New levels are most efficiently created by opening an existing level file, editing that, and saving it as a new level; this eliminates the need to restate sprite size and sprite sheet location each time a level is created. Documentation on the editor can be found at <http://www.mapeditor.org/>.

**FMOD Studio:**

The Spin Engine’s third-party sound system.

**Premake:**

Premake is an automated build tool. It is used with three batch files and one custom lua script, all in the main engine directory: clean.bat, build.bat, and Redo.bat. Clean removes all Visual Studio project files. Build creates all Visual Studio project files. Redo calls Clean and then Build. Premake4.lua contains the project settings relevant to Premake, such as source code and library locations.

**Appendix**

**AntTweakBar Instructions:**

The tweak bar can be toggled between the visible and invisible states by hitting backspace while in Visual Studio’s Debug mode. Click the“save” button at the top of the bar to save all current settings to settings.json. (This will overwrite the previous settings.json file). Bool values are simple check boxes. The fields of integers and floats can be clicked on and typed into directly. Alternately, click on the field and drag away while holding the left mouse button down to create a draggable wheel; rotating the wheel will increase and decrease the value, and the mouse can be brought closer to or further from the center of the wheel for rougher or finer changes. Tweak Bar can be disabled in settings.json.

**InputPlayback Instructions:**

Input playback can be enabled by setting the “Playback Input” field in settings.json to true. Input recording can be enabled by setting the “Record Input” field in settings.json to true . If both features are enabled, recorded input will be played, and and identical recording file will be created. Input playback is not currently intended for the end user and cannot be accessed through the game itself.

The playback system will automatically open an appropriately-titled playback file for any given level as it is loaded. It looks for these in the same directory as the game executable, under the format “LevelName-Playback.json” (I.E. “Level\_2-Playback.json”). If no file is found, an notification will pop up and pause the game; playback will not commence and the engine will expect normal controller input.

When enabled, the recording system will automatically begin saving to a file named for the current level and time, with the format “LevelName-HourMinute.json” (I.E. “Level\_2-0215.json”). The file must be renamed to the playback system’s standard in order to play it back; the playback system will not recognize any other files. To do this, simply replace the time (“0215” in the above example) with the word “Playback.”

**Tiled Instructions:**

Comprehensive documentation for Tiled (and a download for the editor) can be found at <http://www.mapeditor.org/>.

Level files are located in SpinEngine/Assets/Levels. New level files should be added there. New level files must be added to the “Levels” section of master.json. The filename itself matters only for clarity; the “name” field for the level file in master.json is what will be searched for by the Factory.

**Steps for adding a new sprite:**

1. Place the texture in Assets/Textures. (i.e. Assets/Textures/tex.png)
2. Create an Animation file for it in Assets/Animations. (Copy an existing file to save time.)
   1. statename should be “idle” for single-frame sprites, otherwise arbitrary.
   2. texturepath should point to the texture, and texturename be the same name.
   3. start\_i and end\_i should be the first and last frame of the animation, 0-indexed.
   4. totalframes should be the total number of frames in the animation.
   5. timeperframe should be the time in seconds before moving to a new frame.
3. Append the path of the animation file to the appropriate segment of master.json’s “sprites” field. Position is important, because the order of sprites in the “Tiles” field must line up with the order of sprites in the Tiled spritesheet.
   1. If the sprite is meant to be placed using Tiled (this includes being a distinct animation for a different sprite, i.e. Player\_Run AND Player\_Idle must follow this step), append to the “TILES” segment. (Place the Animation filepath at the end of the TILES segment, right before the “OTHER SPRITES” comment.) If you followed this step, proceed to step 4.
   2. Otherwise (I.E. if the sprite is a wall part, otherwise spawned programmatically, or only used once) append to the “OTHER SPRITES” segment. Or prepend it. The order of these sprites does not matter. If you followed this step, skip step 4.
4. If you added a texture intended for use within Tiled, append the texture to Tiled\_Spritesheet.png. Do this even if the sprite is redundant (again, like Player\_Run when Player\_Idle is already present). The purpose is preserving the order of sprites, so that the IDs Tiled assigns to objects remain valid.
   1. If all spaces are full, expand the image downwards another 512 pixels.
   2. If the texture contains multiple frames, append only the one most representative of the animation. (Tiled\_Spritesheet is solely for the benefit of the editor.)
   3. Don’t stress about positioning if your texture is less than 512x512- a small discrepancy will only show in Tiled, and if there’s no edge overlap then it won’t be a significant concern for viewability.