Modern Szoftverfejlesztési Eszközök

Levente Fodor - mu1g7o

November 15, 2023

Contents

List of Figures

1 Introduction

The paper focuses on the most important aspects of version control in game development by highlighting the main differences between practices applied in this area and version control practices used at more traditional branches of software development. Furthermore, the paper intends to demonstrate the typical workflow by showing the functionalities of the widely used Helix Core™ from company Perforce Software. The paper is structured into four main sections: the first section gives an overview of version control principals and concepts. The second discusses the most important differences between software development and game development and how these translate into practice when it comes to version control. The following final sections showcase the Helix Core™ version control software by building a very basic 2D game using the open-source game engine Godot Engine.

2 Overview of version control concepts

The main goals of version control systems can be summarized as follows:

- keep track of change history. By keeping the history of changes on the code base, it becomes possible to revert to a previous state. Furthermore, the identification of bugs' first occurrence and the changeset that introduced them will also be possible.
- maintain several version of the code, e.g. **production**, **release candidate**, **development**. This is particulary useful when some features have already been developed but for some reasons they are planned to be released later in the future.
- allow several people/teams working on the same code base simultaneously. Currently, any commercial software product requires the effective collaboration of many programmers (even hundreds), it would become swiftly chaotic even impossible without a tool that can handle code changes from so many sources.

To satisfy the above mentioned requirements, version control applications have been developed that differ from each other in some or many aspects.
The most straightforward distinction between version control systems is based on whether they treat code repository in a centralized or distributed manner. The former designates an authorative data store and all activies/changes are aligned with reference to this central hub. In the latter case, however, there is no authorative central repository. Code repositories are aligned by activities called merges when changes are combined creating a new repository version incorporating all modifications of the code base.
Centralized version control system (CVCS) usually implements the following workflow [8]:

- check out code repository from central location (usually from server) with all the changes made by other colleagues
- implement own changes
- commit changes, in other words, upload own changes so that others can work with the most up to date version of the repository.

DVCS works with the following typical workflow:

- clone repository with full history of changes
- make own changes
- commit changes to local repository
- push changes to DVCS and merge to main branch related to the activities

Main risk of centralized VCS is related to the authorative repository. If the hosting environment of this repository is unavailable for whatever reasons, it can jeopardize productivity or even cause data loss. The distributed architecture is less prone to such events as it keeps the whole code base at each machine that checks out the repository.

2.1 Branching strategy

Another important aspect of version control is the applied method to keep and maintain different versions of the code base. As mentioned previously, different code versions/branches can be summarized as follows:

- **main** (production): released version of the code, currently in production or in live environment
- **release branch/release candidate**: in a periodic development cycle, there is usually an event, the so-called code freeze, after which a release branch is created that already contains all incremental changes of the software that are planned to go-live with upcoming release. All other changes, that are not part of next release but already in progress stay in the development branch. After the release, the release branch can be archived. There can also be situations when issues are identified after go-live and a hotfix needs to be issued using a **hotfix** branch.
- **development** branch: all changes done by developers start in a development phase (development branch) and mature until they are tested and become ready to go-live (or get cancelled and removed from the repository). Depending on the branching strategy and development situation, there can be one or several development branches, including **feature** branches that are specific to a new functionality and represents a more extensive workload before it can be released.

An important part of development workflow is the method used to create and organize these branches, execute merges. These activites are governed by the so-called branching strategy. There are several branching strategies that can be adopted, the final decision always depends on the nature of development project, available tools (e.g. ticketing system) etc. Further details can be found about Git branching strategy and workflow at [2], or Microsoft's practice at [9], whereas Perforce's tool - which will be demonstrated in later sections - at [6].

Continuous Integration/Continuous Deployment Finally, it is worth mentioning another important terminology that heavily impacts modern software development industry and also can be considered as a broader aspect to the branching strategy. This is CI/CD, used to automate stages of development as much as possible, including testing and bug-fixing. The point of the process is ensuring compatibility between development cycle and operations process. Details of CI/CD principles are beyond of this paper but extensive sources and literature are available, e.g. [10]

3 Video game development as software development

Successful software Game development is a branch of software development, however, many argue that it is a separate genre due to its completely different working pipelines and different composition of development teams, let alone the challenges related to project funding. A nicely written code, good functionality and user experience, fulfilling business specifications and remaininig within budget are usually enough for any commercial software to be successful on the market. In case of game development there are other factors that could potentially undermine the process leading to commercial failures.

Successful video game In fact, the end product of game development is mostly an entertainment or educational software. Apart from the code base that is basically the essence of any software product, the success of video games requires artistic assets, gameplay elements and also an immersive storyline or world creation fitting seamlessly together. These components bring game development closer to creative industries like filmmaking. Actually, it is quite easy to mess up any of the previously described elements or the interactions between them. A video game aiming to be a top-tier product cannot have good controls but horrible graphics or beautiful graphics but nasty gameplay dynamics. Additionally, a good story helps to retain users so that they do not turn away from the game too soon. On the other hand, there are many games considered as successful without nice graphics or story, simply because they are "fun" to play. There is no formal way

to define "fun", hence it becomes a difficult task to define the ultimate definition of "good" or "succesful" video games, just like in case of movies.

Team composition The additional components to be considered in the game development workflow also affects team composition. The traditional software development teams usually comprise the following roles: UI designer, UI developer, front-end and back-end developers, product manager, database and system administrators, dev-ops, quality assurance (testers). Whereas in game development teams other roles show up on top of the previous list, mostly related to artistic work: art teams designing 2D/3D objects (separate developers for game menus and other assets), animation teams, producer, game designer, content writer [7].

Version control in game development The previously mentioned characteristics of video game production and team composition affect also the version controlling activities applied during development. Different teams have different needs and a good VCS can make their interactions smoother by adding as little additional administrative burden as possible while speeding up development cycles. The version control system needs to be able to handle and integrate the work of art teams and game designers into traditional software development workflow. There are several options available when it comes to decide which version control system to use. The below chart shows the distribution of VCS among game developers.

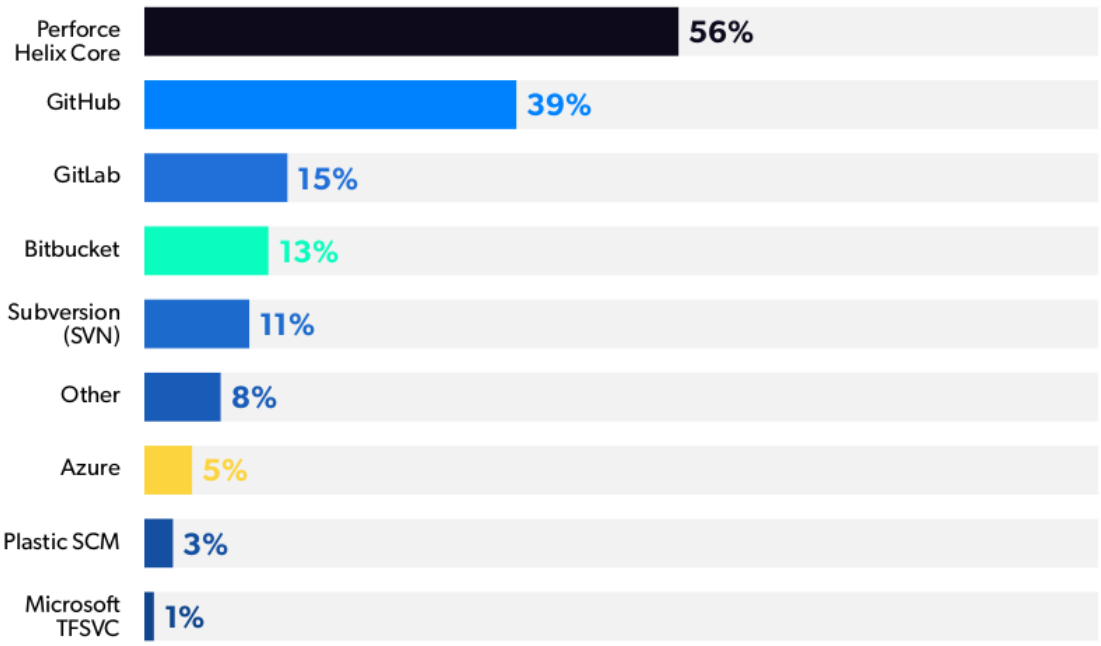

Figure 1: game dev survey vcs

Source: The State of Game Development Report: 2020 & Beyond [1]

After the more theoretical first two sections of the paper, the next sections guide through a simple game development project using the most widespread version control system of the industry, Helix Core™.

4 Version control using Helix Core™

In this section, a very basic 2D game is created with the game engine Godot. The development process is version controlled using the Helix Core application to demonstrate the major steps and show the differences compared to traditional SW development workflow. Some basic characteristics and definitions of the application:

- centralized version control system: files are maintained on centralized server

- files are stored in shared repositories called depots on server: typically 1 depot per project
- workspace: mapping of relevant files between local machine and server

4.1 Setup Helix Core

The setup of Helix Core follows the instructions from the Helix Core Server Administrator Guide (or in [4]) relevant for Linux (Ubuntu) systems. The following commands were issued for setting up the version control application, as can be found in the aformentioned documentation.

1. Download perforce public key

```
$ wget https://package.perforce.com/perforce.pubkey
```

2. Obtain the fingerprint of the public key and verify

```
$ gpg -n --import --import-options import-show perforce.pubkey
$ gpg -n --import --import-options import-show perforce.pubkey |
grep -q "E58131C0AEA7B082C6DC4C937123CB760FF18869" && echo "true"
```

3. Add the public key to your keyring

```
$ wget -qO - https://package.perforce.com/perforce.pubkey |
sudo apt-key add -
```

4. Create a new file for the Perforce repository

```
$ sudo nano /etc/apt/sources.list.d/perforce.list
```

5. In the new file, input the following line

```
deb http://package.perforce.com/apt/ubuntu focal release
```

   Make sure the version matches the Linux/Ubuntu version of the machine's system. E.g 'focal' needs to be replaced by 'jammy' if the most recent Linux/Ubuntu LTS version (as of 15-11-2023) is running on the machine.

6. Update machine and install Helix Core

```
$ sudo apt-get update
$ sudo apt-get install helix-p4d
```

7. Run configure file

```
$ sudo /opt/perforce/sbin/configure-helix-p4d.sh
```

   During the installation process, I had problems with this step when I tried to remove and install the helix-p4d package. The correct way of removing all related data is summarised as follows:

   - Remove package

```
$ sudo apt-get remove helix-p4dctl
$ sudo apt-get purge helix-p4dctl
```

- For some reasons, the file /etc/perforce/p4dctl.conf.d/p4d.template was missing after these steps. The file was recreated with below content (downloaded from here and slightly modified) and the install process worked again.

```
p4d %NAME%
{
    Owner    =    perforce
    Execute  =    /opt/perforce/sbin/p4d
    Umask    =    077

    # Enabled by default.
    Enabled  =    true

    Environment
    {
        P4ROOT    =    %ROOT%
        P4SSLDIR  =    ssl
        PATH      =    /bin:/usr/bin:/usr/local/bin:...
        /opt/perforce/bin:/opt/perforce/sbin

    # Enables nightly checkpoint routine
    # This should *not* be considered a complete backup...
    solution
    MAINTENANCE =     true
    }
}
```

8. Download visual interface (p4v) from Download Page. P4v is the client application communicating with the server. Server in this case is created on local machine. In case of reinstall, the P4V client might also need to be unpacked again so that the already created workspaces do not block the execution. For more information, check [5].

9. Additional config steps

```
$ export P4PORT=ssl:1666
$ export P4USER=super
# add the following line to ~/.bashrc
export PATH=${PATH}:~/perforce/p4v-2023.3.2495381/bin
```

10. Login and start p4v

```
$ p4 login
$ p4v # launches GUI
```

Having done the above steps, the GUI can be launched bringing up the screen below. The location of the server (where all shared files/projects are stored) /opt/perforce/servers/master/root as the server is also created on local machine. This folder is used to create the mapping between server and other client machines and workspaces.

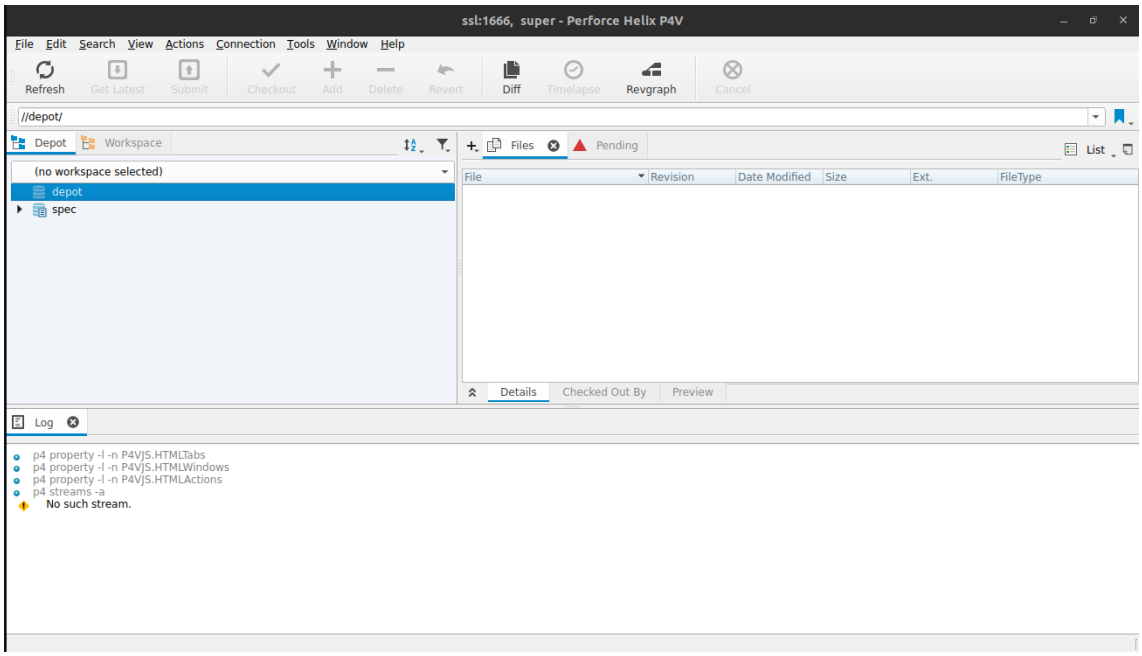

Figure 2: p4v login

4.2 Setup Godot

Setting up Godot game engine is relatively easy compared to other game engines like Unity™ or Unreal Engine™. Godot is lightweight and there are basically two options to get hold of the application:

- Download pre-build application specific to host machine OS and programming language (GDScript or .Net) Download page
- Build from source following instructions Build from source.

For the purpose of this paper the first approach was adopted by downloading the Godot 4.1.2 stable version for Linux. After downloading and unpacking, Godot is ready to use:

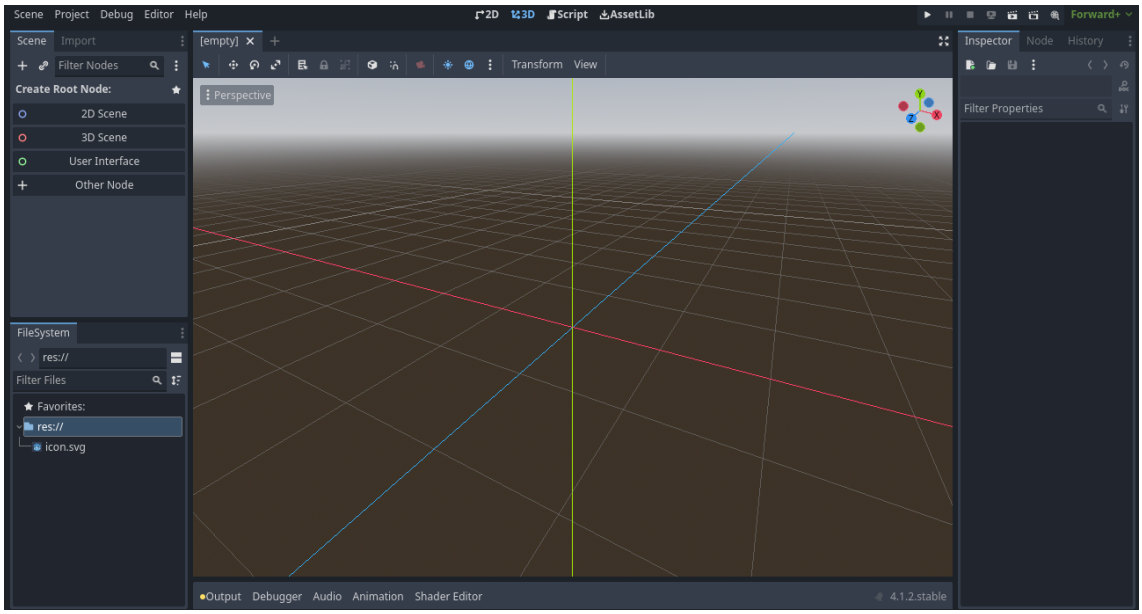

Figure 3: godot editor

The Godot Engine documentation including tutorials can be found in [3].

4.3 Setup version control for development project

The final step before the start of the development is to prepare the server depot and local workspace areas, connect and initialize them by adding the first version of files from the Godot project. The following section details the development steps of a very simple 2D game based on instructions from the Godot tutorial at Godot 2D game instructions using Perforce's Helix Core as version control system.

1. First, start by setting up a new depot on the server. Login to perforce. Run from terminal:

```
$ p4 login
$ p4v
```

2. Choose Tools => Administration (or press CTRL+SHIFT+A). This brings up Helix Admin.

3. From Helix Admin, File => New => Depot. Enter a name for the depot (*godot2d* in my case), choose stream as depot type then press OK. The new empty depot is created on the server. Finally, close Helix Admin.

4. Create TypeMap. TypeMap tells Perforce how to treate different type of files and what rules should be applied on the type of files when being edited. E.g. some binary files cannot be modified if someone else is also working on it as conflicts cannot be resolved and in such cases either one or the other version of the file has to be retained and the other will be dropped. For these files, it is better to use the *+l* modifier which automatically locks the file so multiple users are not allowed to edit the file at the same time.

```
TypeMap:
    binary //....meta
    binary+wS //....exe
    +l //project_a/....obj
```

   The '...' symbol is a recursive wildcard in Perforce ('*' is non-recursive wildcard). In this example, all files ending with .meta should be treated as binary, the .exe files should be treated as binary with the 'w' and 'S' modifiers (always writable and using latest revision), whereas all .obj files in the project_a folder should be locked. To create a TypeMap, open a terminal and type the following command:

```
$ p4 typemap
```

   The command brings up a typemap template that is already defined for several file types. Unfortunately, the default editor for p4v is not the most user-friendly, so it makes sense to switch the editor before calling the previous command. To switch the editor to nano, type:

```
$ p4 set P4EDITOR=nano
```

   Now, it is much easier to make changes to the template typemap. The following typemap will be used for the project at hand:

```
Typemap:
    binary+lS //....png
    text+lS //....import
    text+w //....gd
    text+w //....tscn
    binary //....godot
```

5. Next up, a stream is to be created. Streams are useful assets in Perforce as they allow different teams to work on the project without distracting each other. E.g. 3D modellers can work on one stream and developers on another one and once they are both happy with the outcome, the development stream can be merged to the 3D modellers' stream to make the changes available to them. For the time being, only one stream is created. To do this, click the Add tab button on the right pane and select Stream Graph. Then right click in any blank space and select New stream. Populate the input fields:
   Stream name: main
   Stream type: mainline
   Depot: godot2d
   Uncheck Create Workspace and Populate checkbox to avoid these automatic steps. These will be executed manually.

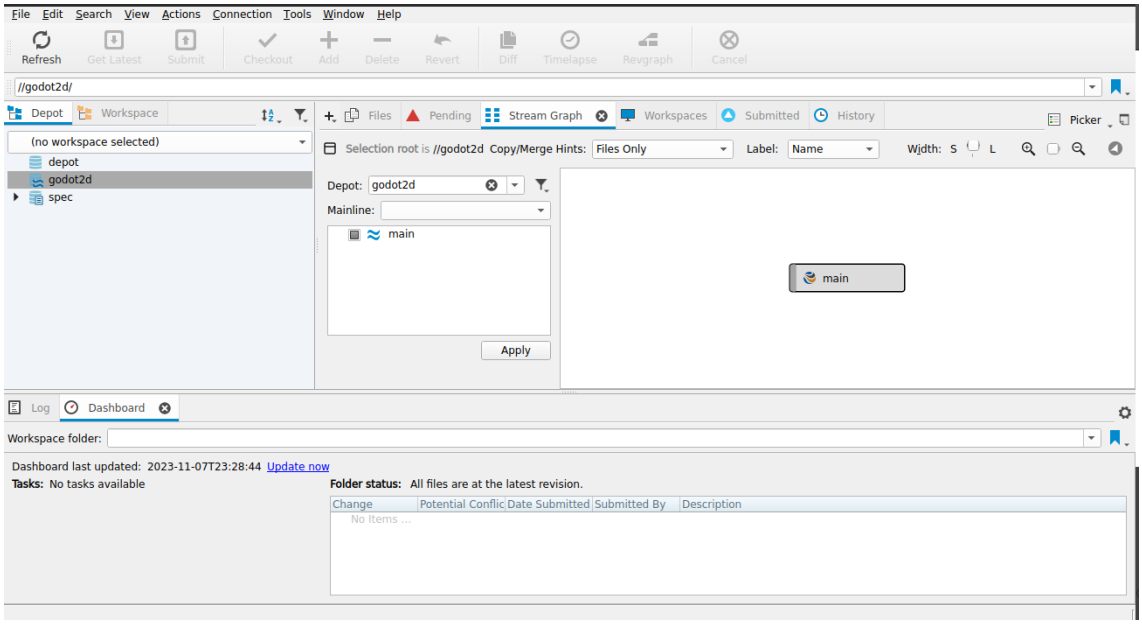

Figure 4: new stream

6. Next, a new workspace has to be created in p4v: click View => Workspaces, right click on blank area on the right side when the Workspaces tab is active, click on new workspace. Instead of clicking, just press CTRL+N.

7. Workspace name and root need to be given. The former usually follows the naming convention of *<user>_<machine_name>_<project_name>* which becomes in my case *super_lefodor-HP_godot2d*. The latter is set to any location on the local machine that ensures convenient working with the repository. Additionally, the stream is also to be filled out.

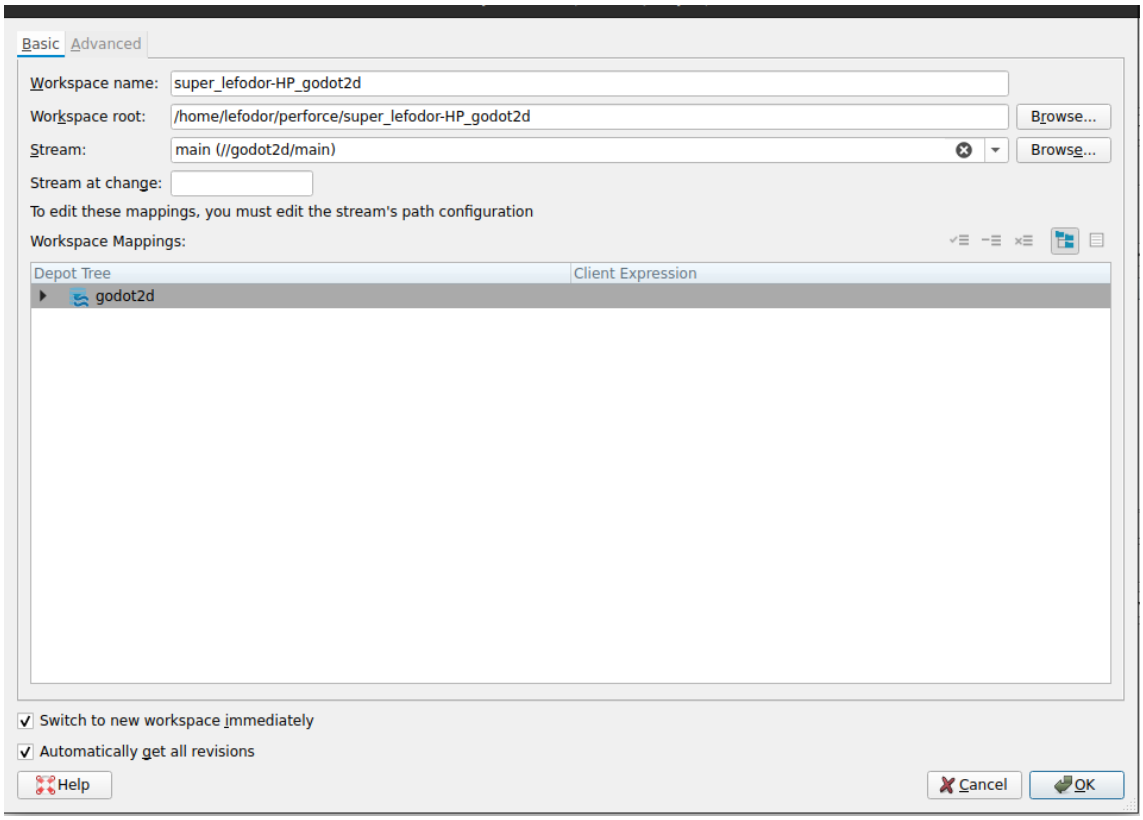

Figure 5: new-workspace

Right click on all depots that are not required in the repository => click Clear. After this step, only depot godot2d should have green pipe next to it. Then click on OK to create new workspace.

8. Similarly to other version control systems, it is also possible to define a file that specifies files that are not subject to version control. In a new terminal window:

```
$ p4 set P4IGNORE=.p4ignore
$ touch /home/lefodor/perforce/super_lefodor-HP_godot2d/.p4ignore
```

In P4V, on the left pane right click on the file and click **Mark for Add** to add it to the default changelist, and then hit **Submit** in the upper button bar while the file is selected.

9. Add files to workspace. Copy blank Godot project folder to workspace root folder. Refresh P4V if needed to see the newly copied files in the left pane. Add the files to the default changelist and hit **Submit**. Only the non-hidden files have been added to the workspace for version controll (note the small green icon next to files).

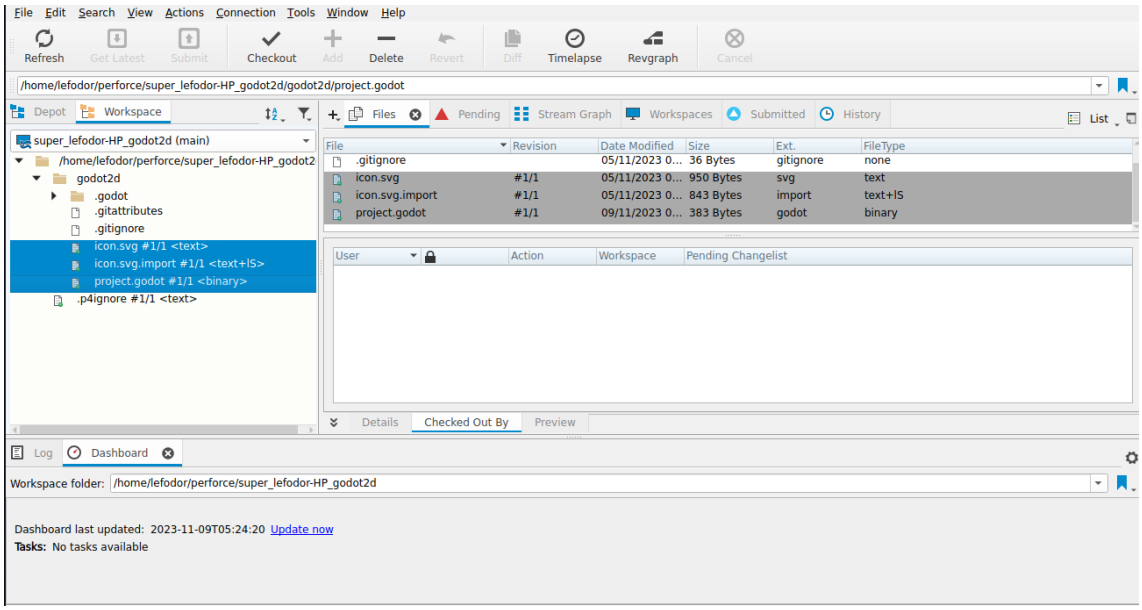

Figure 6: init-workspace

Having done the final step, the folder is ready to be shared and version controlled via the server depot and the local workspace contents, currently 3 files are part of the repository of *godot2d*.

4.4 First steps of game development

Now, that the version control system, the game engine and the connections between the two are set up, the development project can finally kick off. The steps in this sections are following the guide Godot 2D game instructions.

1. Download assets and add them to version controll. Files are downloaded from here. There are additional two more folders called *art* and *fonts* extracted and added to project folder. They need to be added to version control by selecting them, click **Add** button and then **Submit**.

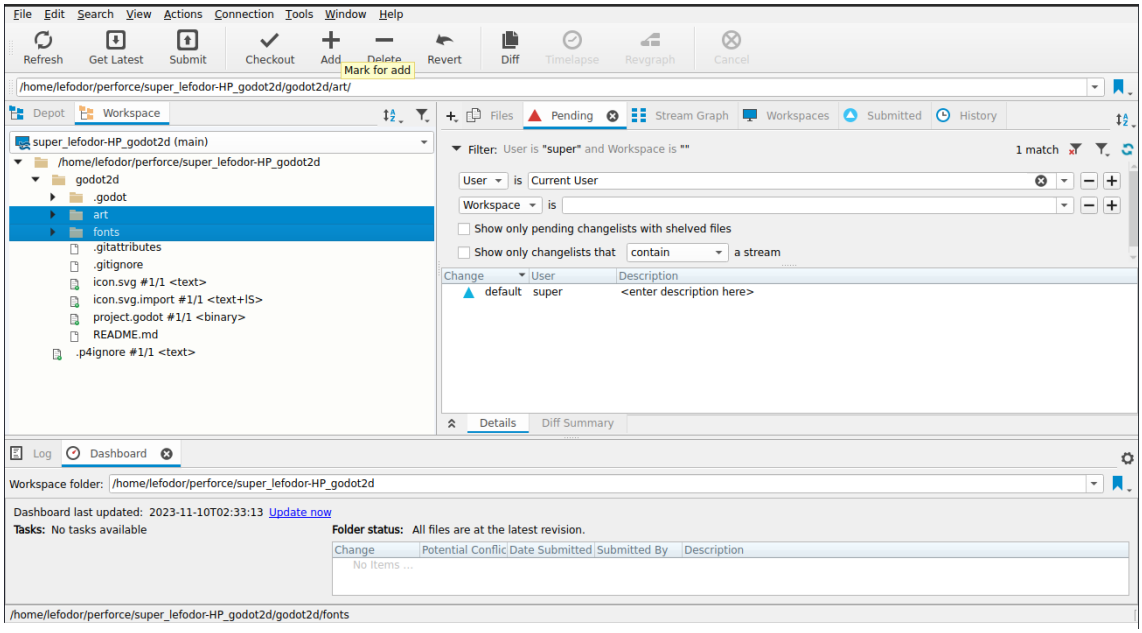

Figure 7: add assets to VC

2. Changes to *.godot* project file

- Change project settings: click **Project** => **Project Settings**. Under **Display/Window** change Viewport Width and Height to 480 and 720 respectively in the Size section and set Mode to *canvas_items* in Stretch section. In Perforce, the following changes can be observed: when *godot2d* folder is checked out all the files are added to the changelist. When the changelist is submitted, new revision record shows up on the right side pane, under History button. With right click on the revision, list of changed files can be opened (by clicking on *View submitted changelist*)

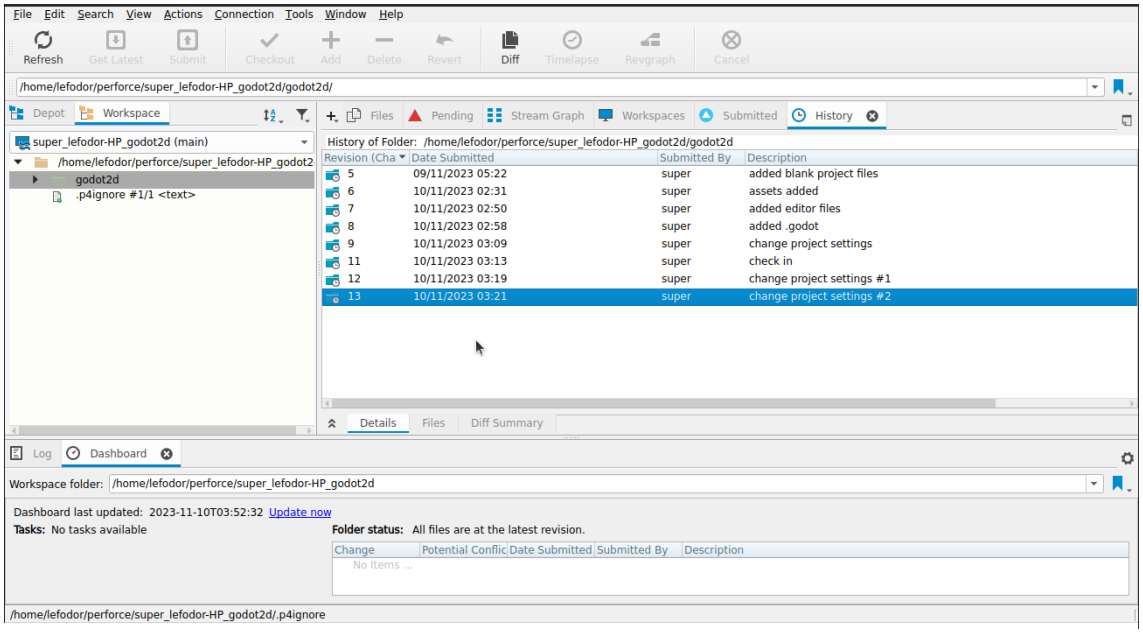

Figure 8: project file changes in perforce

3. Create player scene The player scene contains all objects related to the character controlled by the player.

- Click on **Other node** button and add an *Area2D* node, rename it to player and save as *player.tscn*.

- Add sprite to player. Add a child node *AnimatedSprite2D* to the player node and click the sprite frame property under *Animation* section. Here, we can create 2 animations, the first is called 'up', and add animation frames *playerGrey_up1.png* and *playerGrey_up2.png* from the downloaded asset folder. Secondly, add another animation called 'walk' with frames *playerGrey_walk1.png* and *playerGrey_walk2.png*. Rescale by setting both x and y parameters under *Node2D/Transform* to 0.5.

- Add collision shape. Add a *CollisionShape2D* as child node to player node. This will make possible for the character to collide with other objects. Select *CapsuleShape2D* for shape property under *CollisionShape2D* at the right hand-side pane of the screen and adapt the size of the capsule on the graphical interface so that it covers the animated sprite.

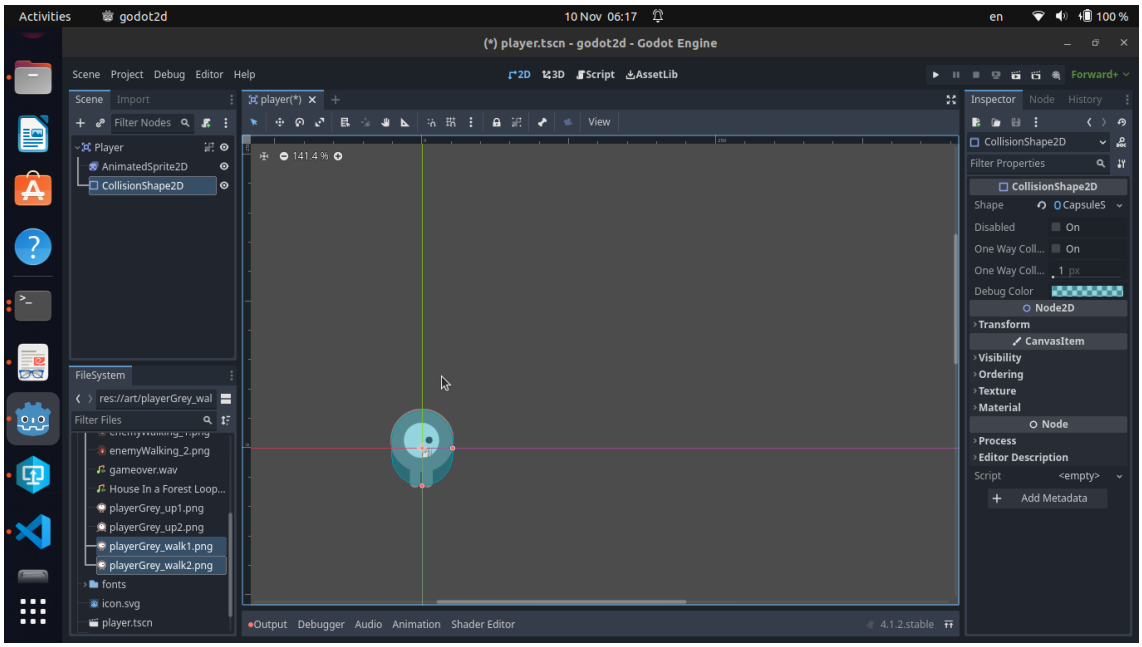

Figure 9: adding collision shape to player

- After submitting in P4V, the changelist looks as follows for the above changes. Do not forget to add the new *player.tscn* file to the version controlled workspace.

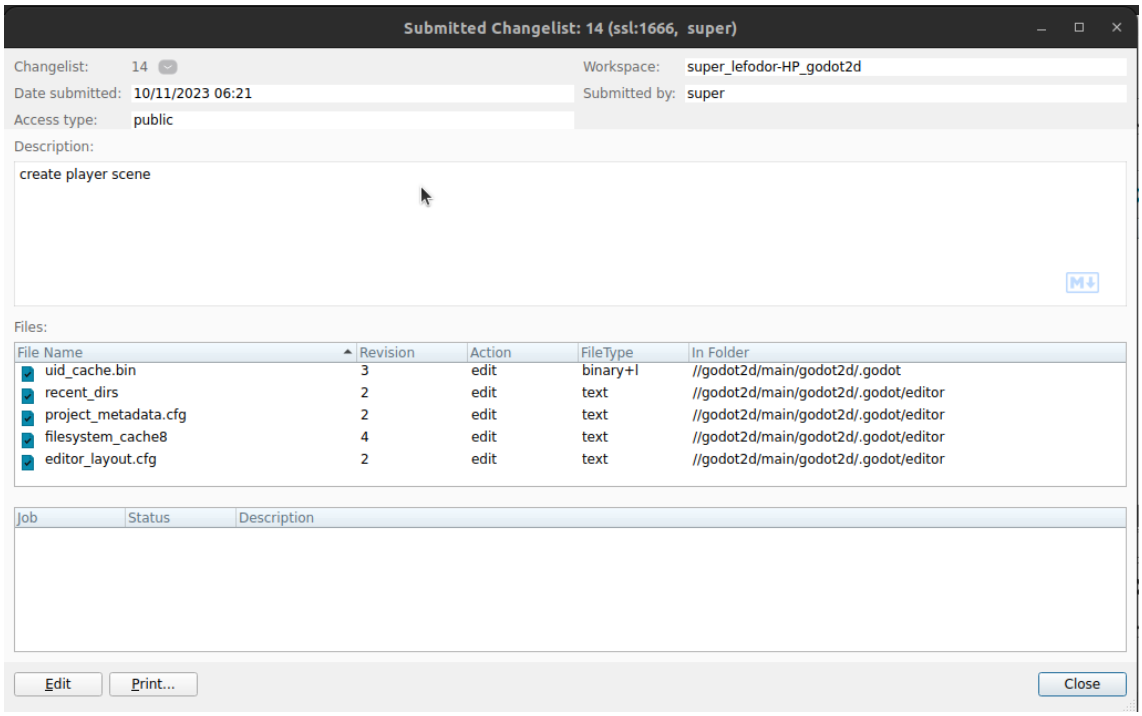

Figure 10: p4v for create-player-scene

4.5 Improve workflow with branching

Up to this point, only some basic activities of the version control software have been demonstrated like saving changes to the project files and displaying history of changes. No branching has been done, however, it is more important feature when it comes to complex development project, faster iteration cycles and CI/CD principles. In this section, a new development branch (stream in Perforce's terminology) is created beside the one called main and all incremental development is done to the development branch which is later merged to the main one.

4.5.1 Setup new stream

In order to setup a new stream in Perforce, open the **Stream Graph** tab and right click on the blank area, then choose *New Stream*. Configure the new stream based on the pop-up window as displayed below.

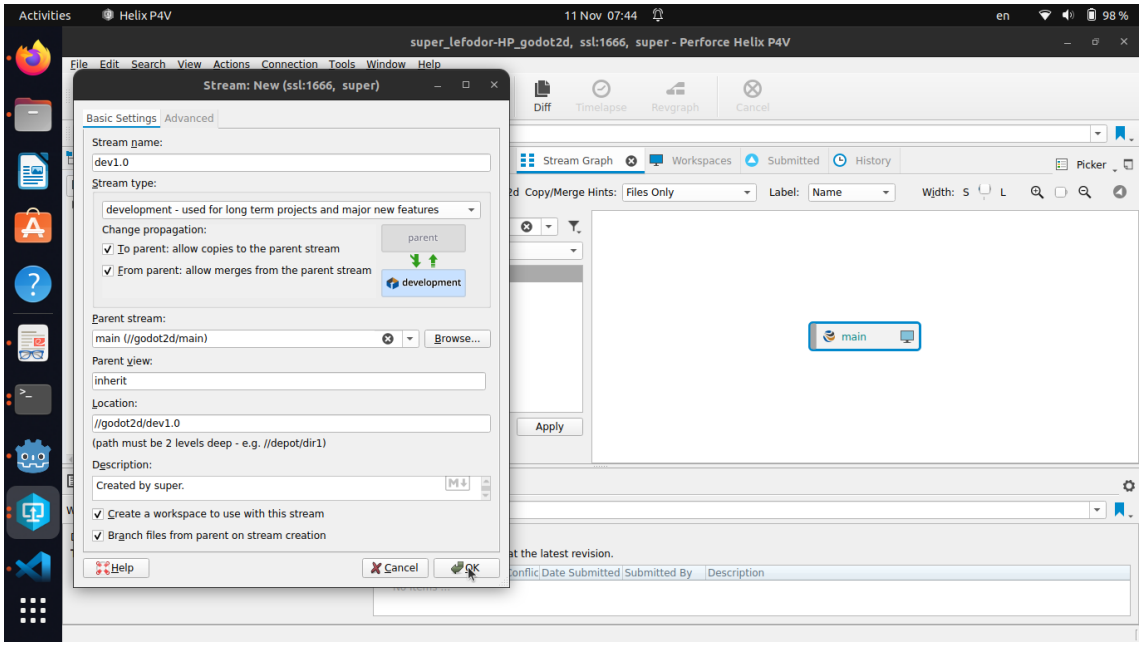

Figure 11: p4v new stream for branching

A new development stream (branch) has been created. Notice that a new workspace area is also assigned:

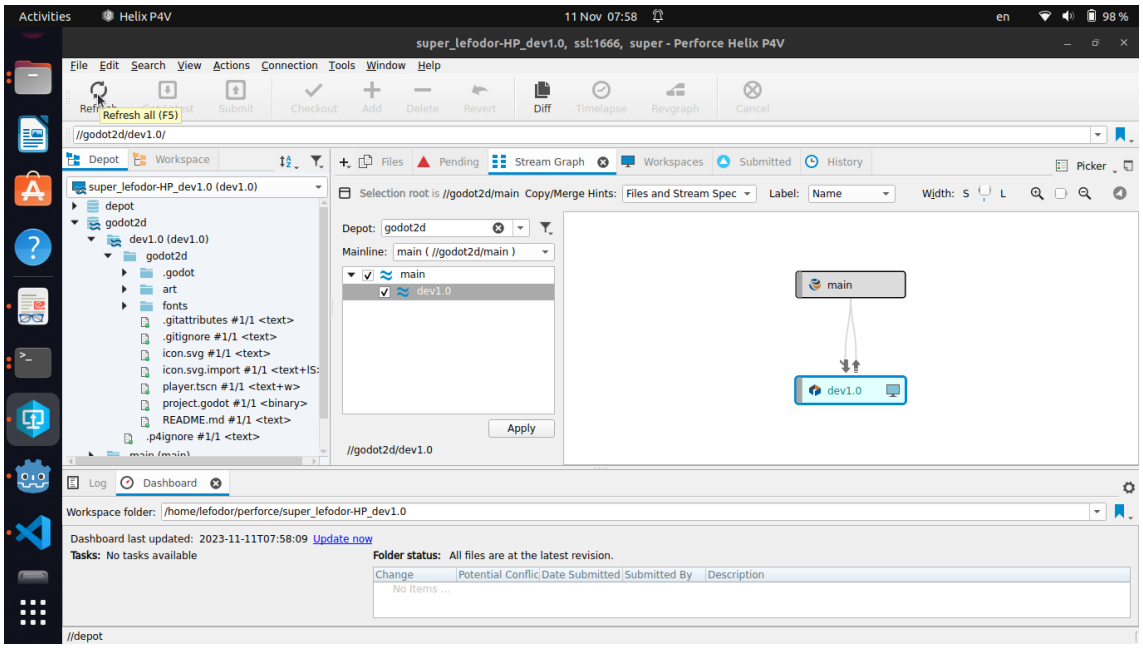

Figure 12: new stream (branch) dev1.0

From this point onwards, all changes are to be made on the development stream and later merged back to the more stable main stream. Also, changes need to be made to the workspace related to this branch and not on the one linked to main.

4.6 Coding the player scene and creating the enemy

Coding player scene. Add script to player by selecting the scene and click on the "Attach Script" button.

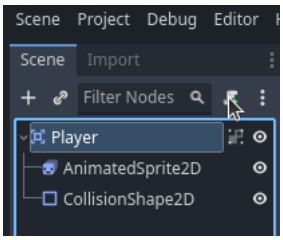

Figure 13: add script to player

The added script (*player.gd*) should look like as below:

```
extends Area2D

signal hit

@export var speed = 400 # How fast the player will move (pixels/sec).
var screen_size # Size of the game window.

# Called when the node enters the scene tree for the first time.
func _ready():
  screen_size = get_viewport_rect().size


# Called every frame. 'delta' is the elapsed time since the previous frame.
func _process(delta):
  var velocity = Vector2.ZERO # The player's movement vector.
  if Input.is_action_pressed("ui_right"):
    velocity.x += 1
  if Input.is_action_pressed("ui_left"):
    velocity.x -= 1
  if Input.is_action_pressed("ui_down"):
    velocity.y += 1
  if Input.is_action_pressed("ui_up"):
    velocity.y -= 1

  if velocity.length() > 0:
    velocity = velocity.normalized() * speed
    $AnimatedSprite2D.play()
  else:
    $AnimatedSprite2D.stop()

  if velocity.x != 0:
    $AnimatedSprite2D.animation = "walk"
    $AnimatedSprite2D.flip_v = false
    # See the note below about boolean assignment.
    $AnimatedSprite2D.flip_h = velocity.x < 0
  elif velocity.y != 0:
    $AnimatedSprite2D.animation = "up"
    $AnimatedSprite2D.flip_v = velocity.y > 0

  position += velocity * delta
  position = position.clamp(Vector2.ZERO, screen_size)
```

```
func _on_body_entered(_body):
  hide() # Player disappears after being hit.
  hit.emit()
  # Must be deferred as we can't change physics properties on a physics callback.
  $CollisionShape2D.set_deferred("disabled", true)

func start(pos):
  position = pos
  show()
  $CollisionShape2D.disabled = false
```

After this step, it is already possible to run the game by pressing F6 and use the arrow buttons to control the player's character up/down and left/right. Final step of this point is the preparation for collisions. The *signal hit* line generates a so called signal. Signals are emitted upon certain events, in this case, when the player gets hit.

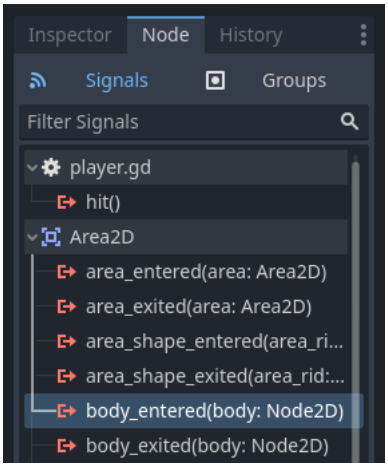

Figure 14: player signals

The *body_entered* signal is connected to the player meaning that if an Area2D object hits the player the signal is emitted. By connecting the signal, a function is created (*func _on_body_entered*) in the script file that defines the behaviour upon signal emission. The final piece of code (*func start()*) defines the starting position of the player when the game starts. The next step in the version control is to merge changes in stream *dev1.0* back to the *main* stream. Currently, the Stream Graph in P4V looks like this.

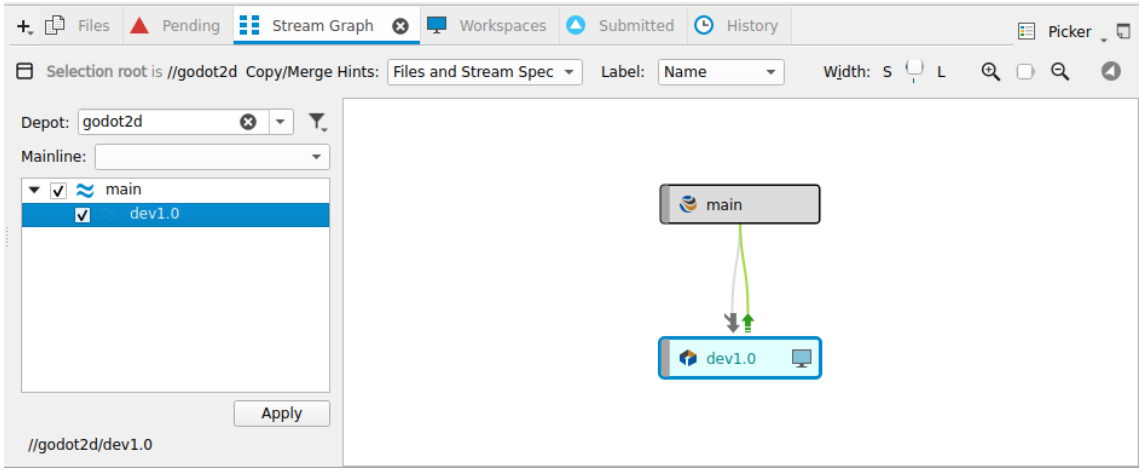

Figure 15: copy files from dev to main

The green arrow means merge or copy operation is available. As for the terminology, "merge" means send changes from the more stable stream (main) to the less stable one (dev) and "copy" means vica versa. This case, only the less stable stream *dev* stream is changed, so only copy operation is needed. Gray arrow means no copy or merge operations are necessary. Right click on the target stream => *Copy files to main*. In the popup window, the workspace for the target stream has to be activated then click on *Copy* to copy the files. Finally, in the main stream, the changes from the copy operation has to be submitted. With these steps, changes from the development branch are also saved in the main branch and branches are synchronised.

Creating the enemy Create a new scene called "Mob", which will be instanced as many times as wished in random locations of the screen and move in randomly selected directions. Click **Scene => New Scene** and add the following structure:

- RigidBody2D (called Mob)
  - AnimatedSprite2D
  - CollisionShape2D
  - VisibleOnScreenNotifier2D

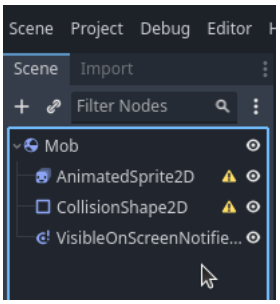

Figure 16: create enemy

Select the Mob scene and set the to *Gravity scale* in the *Inspector* tab to 0. In section *CollisionObject2D*, under **Collision**, in property **Mask** uncheck the 1. Setup the animation for the enemy similarly as in 3. There are three animations to be defined: *fly*, *swim* and *walk*. Drag the corresponding sprites for each of them, set the FPS to 3. In the *Inspector* window, under **Node2D/Transform**, set *Rotation* to 90 and *Scale* to 0.75.

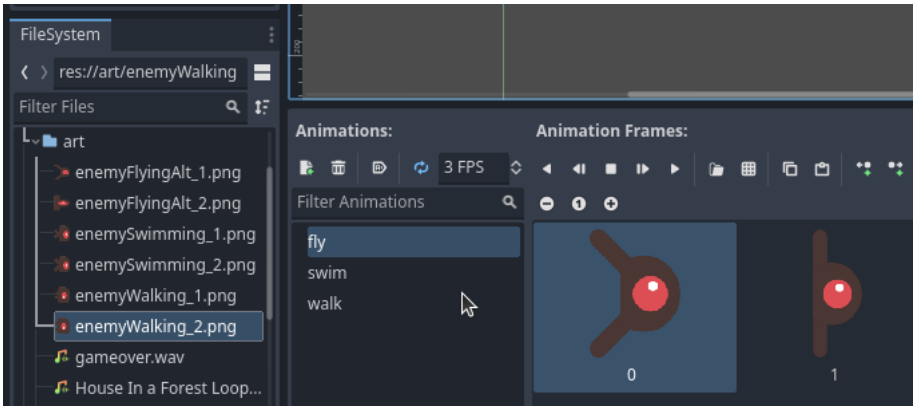

Figure 17: animation enemy

Finally, define *CapsuleShape* for the collision shape's *Shape* property. Add a ew script to the Mob scene with the below content and save the script as *mob.gd*:

```
extends RigidBody2D
```

```
# Called when the node enters the scene tree for the first time.
func _ready():
var mob_types = $AnimatedSprite2D.sprite_frames.get_animation_names()
$AnimatedSprite2D.play(mob_types[randi() % mob_types.size()])


# Called every frame. 'delta' is the elapsed time since the previous frame.
func _process(_delta):
pass

func _on_visible_on_screen_enabler_2d_screen_exited():
queue_free()
```

Regarding P4V, same steps have to be done as previously when player scene and script were added (add new files, submit changes and copy files from dev stream if sync is needed at this moment). The *dev* workspace looks as follows after creating and adding the enemy.

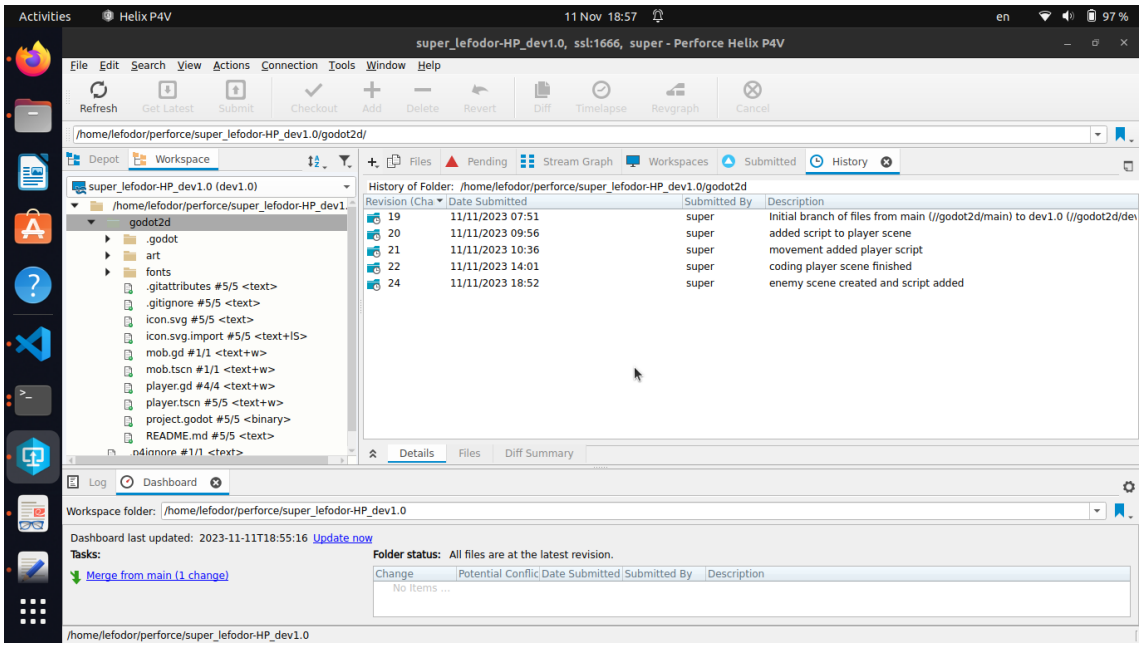

Figure 18: workspace after enemy creation

4.7 Bring everything together

The main game scene. Add main scene containing player and mob scenes. Additionally, this scene controlls the flow of the game by handling timers. Create new scene by adding a *Node* called **Main**. Then click on the **Instance** button (see cursor on the image) and select *player.tscn*.

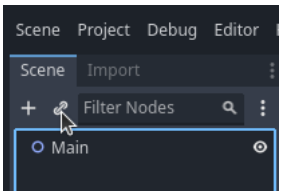

Figure 19: workspace after enemy creation

Add three Timer objects to *Main* and name them and set *Wait Time* property as follows:

- MobTimer - control mobs spawn: 0.5

- ScoreTimer - increment score by seconds: 1
- StartTimer - delay before starting: 2

Additionally, create a *Marker2D* for the starting position of the player and set its *Position* property to 240, 450.

4.7.1 Spawning mobs

Enemies should be created randomly on the edge of the screen. To achieve this, add a *Path2D* node and call it **MobPath** as child of **Main**. Select the 4 corners of the screen to setup the path of the curve used for spawning enemy objects. Finally, a *PathFollow2D* node needs to be added as a child of **MobPath**, name it **MobSpawnLocation**. The scene is expected to look as below with the created curve:

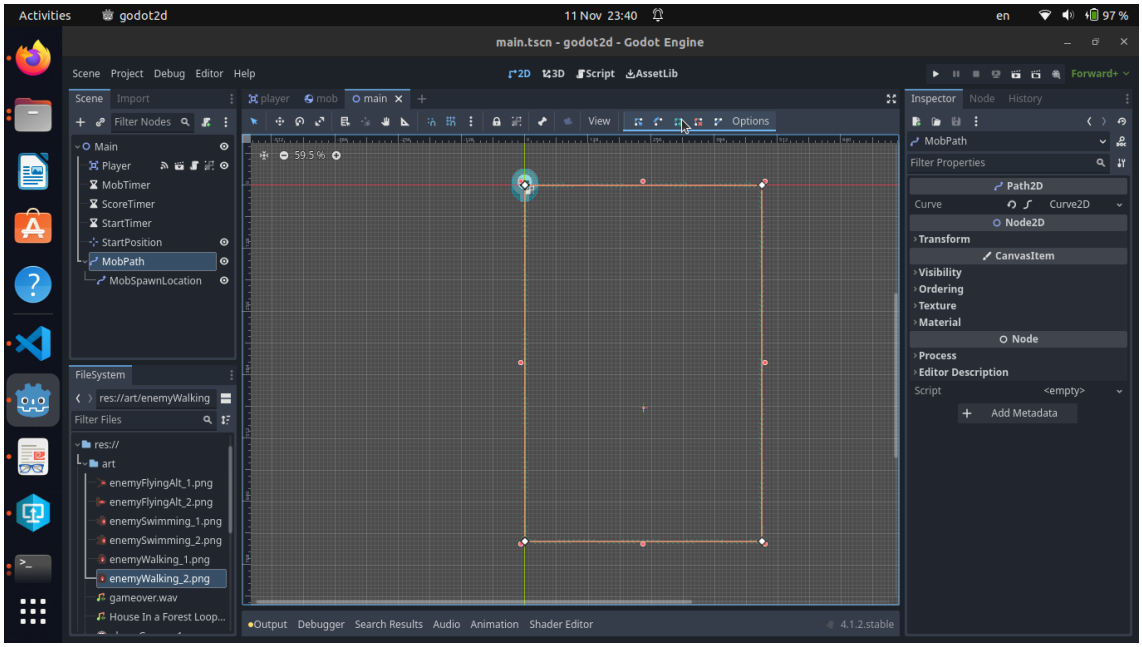

Figure 20: mob path

Add a script to the main scene with the following content:

```
extends Node

@export var mob_scene: PackedScene
var score

# Called when the node enters the scene tree for the first time.
func _ready():
    pass #new_game()


# Called every frame. 'delta' is the elapsed time since the previous frame.
func _process(delta):
    pass


func game_over():
    $ScoreTimer.stop()
    $MobTimer.stop()

func new_game():
```

```
    score = 0
    $Player.start($StartPosition.position)
    $StartTimer.start()

func _on_start_timer_timeout():
    $MobTimer.start()
    $ScoreTimer.start()

func _on_score_timer_timeout():
    score += 1

func _on_mob_timer_timeout():
    # Create a new instance of the Mob scene.
    var mob = mob_scene.instantiate()

    # Choose a random location on Path2D.
    var mob_spawn_location = get_node("MobPath/MobSpawnLocation")
    mob_spawn_location.progress_ratio = randf()

    # Set the mob's direction perpendicular to the path direction.
    var direction = mob_spawn_location.rotation + PI / 2

    # Set the mob's position to a random location.
    mob.position = mob_spawn_location.position

    # Add some randomness to the direction.
    direction += randf_range(-PI / 4, PI / 4)
    mob.rotation = direction

    # Choose the velocity for the mob.
    var velocity = Vector2(randf_range(150.0, 250.0), 0.0)
    mob.linear_velocity = velocity.rotated(direction)

    # Spawn the mob by adding it to the Main scene.
    add_child(mob)
```

The *Main* node needs to be able to access variables from *Mob* scene. For this, the *Mob Scene* <empty> button in the *Inspector* tab and load *mob.tscn* in the pop-up window.

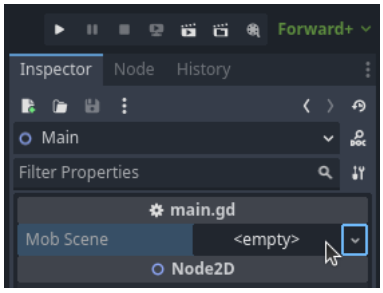

Figure 21: mob scene variables

Sort out signals:

- Connect the *hit* signal of the player scene with a receiver method called **game_over**.
- Connect *timeout()* signal of each Timer nodes to the main script.

Now, pressing F5 should run the game with a player character disappearing when being hit by an enemy.

4.8 Version control of final steps

Having done all the steps above, commit changes to the *dev1.0* stream, the version control system shows the below picture:

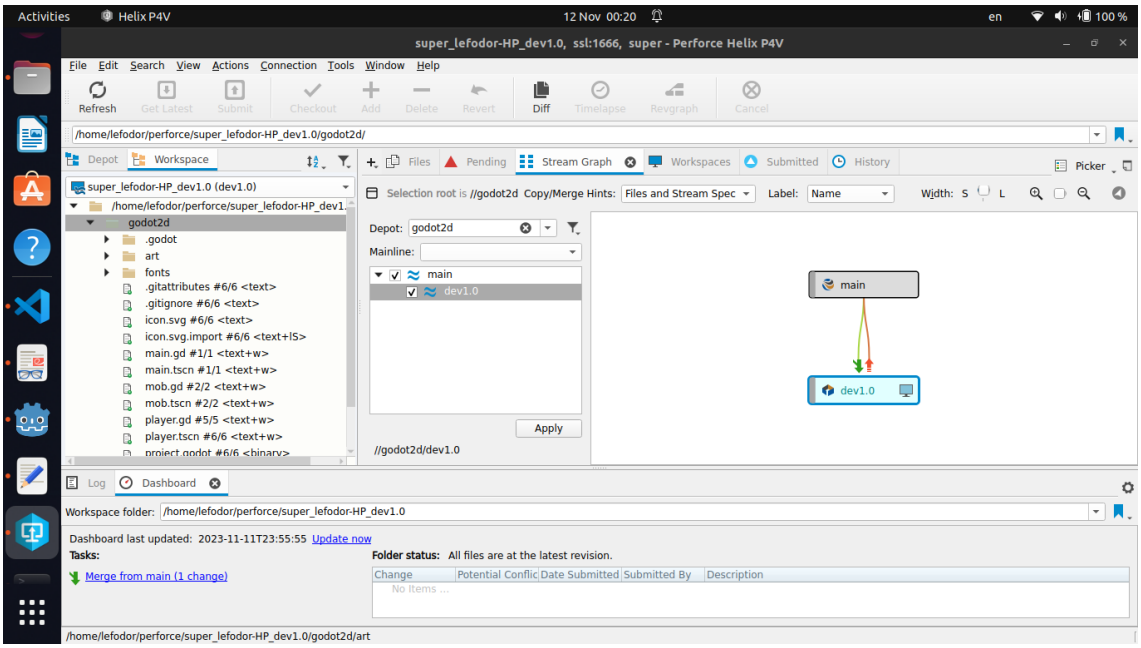

Figure 22: copy merge main

The orange arrow means there is a need to merge changes from the *main* stream and only after this becomes possible to copy files from the development stream back to main.
Proceeding with the merge (downstream synchronization from more stable (main) stream towards less stable (development) stream), the below conflict is shown:

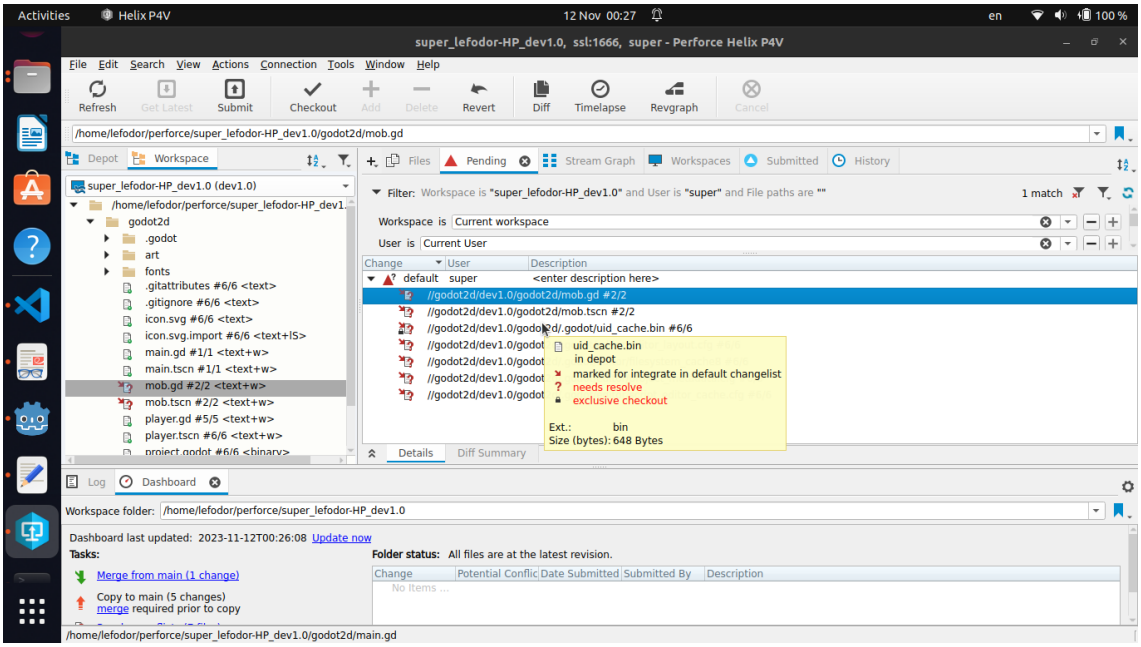

Figure 23: show main dev conflict

The conflicts are needed to be resolved manually to make sure the correct version is retained. The *Resolve* pop-up window shows the possibilities to Accept Source/Target/Merged or Run Merge

Tool. For all conflicts, the 'Accept Target' option was selected as only the *dev1.0* stream was updated during development process, different versions in the *main* stream may only occur due to autosave method of the game engine. If not only one developer works on the project then the merge/copy history has to be checked and select the proper resolve method accordingly.

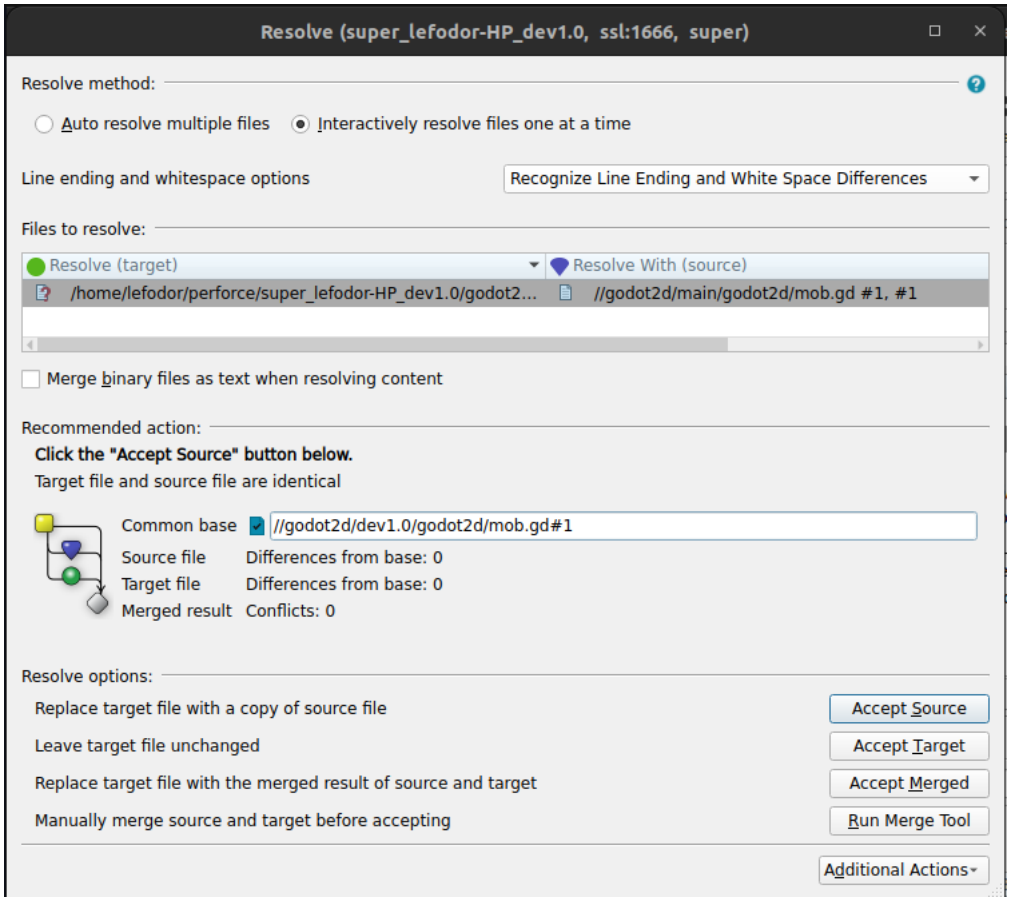

Figure 24: resolve main dev conflict

After resolving the conflicts, the copy method from the *dev1.0* stream is possible (green arrow).

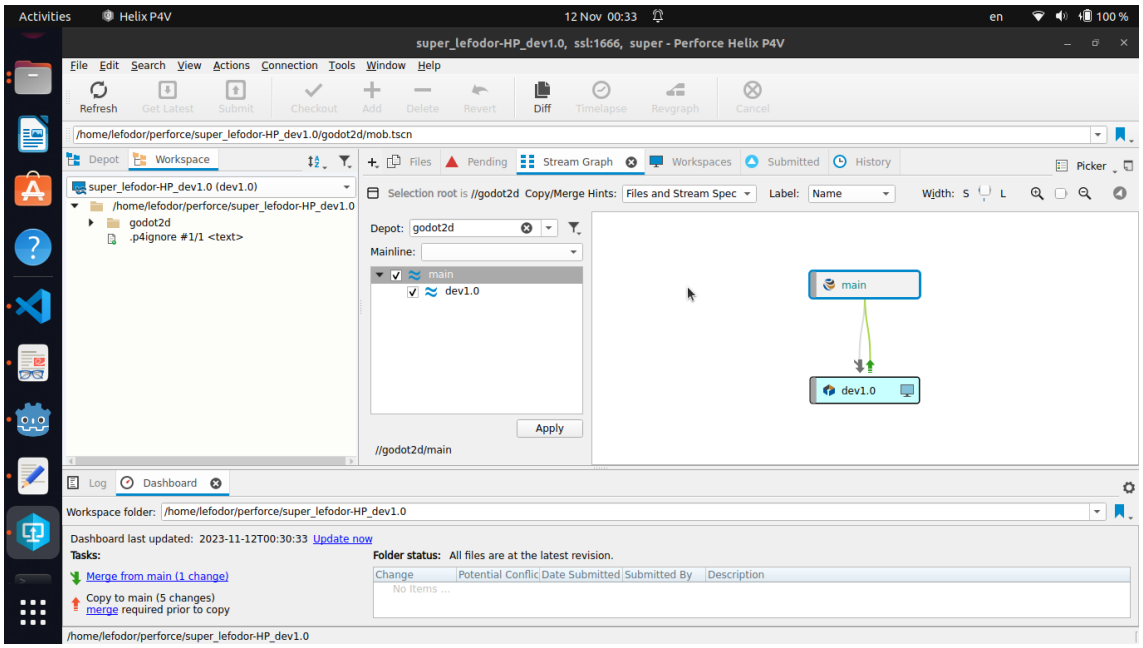

Figure 25: after merge conflicts resolved

As a last step, we can observe the change history for both *main* and *dev1.0* streams

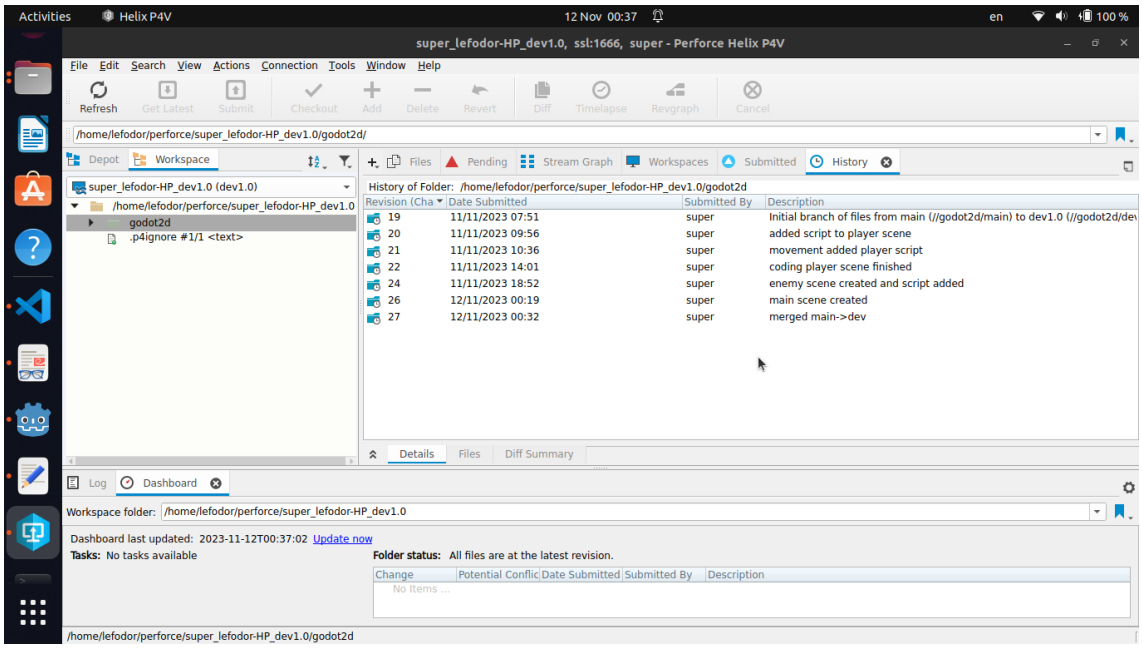

Figure 26: final dev1 history

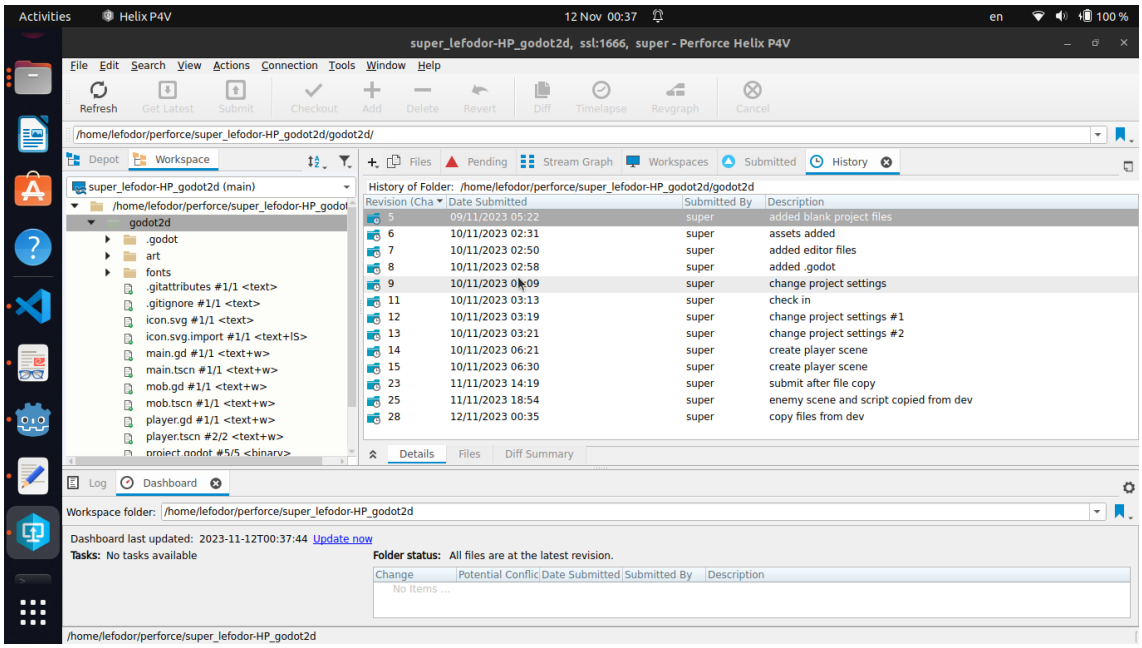

Figure 27: final main history

5 Version control of artistic assets

At this point, the game is already working with the most important features implemented. Following the guide, there are two additional features contributing to the experience, namely the heads-up display and sound effects, however, the purpose of this paper is to demonstrate version control application **Helix Core**™ which is already fulfilled. There remains only one outstanding point, the version control of animations and other artistic assets used for the game. This scenario plays important role when the game development includes also production of own assets and art teams are also working on animations and 3D models.

For demonstration purposes, the sprites used for the game will be amended and see what version control techniques are available in **Helix Core**™ to keep track of change.

- create new stream for art team => call this *art1.0*
- observe how the development branches (streams) interact with the main branch, e.g. changes in artistic elements copied to main and from there, merged to the other development stream used by game programmers.

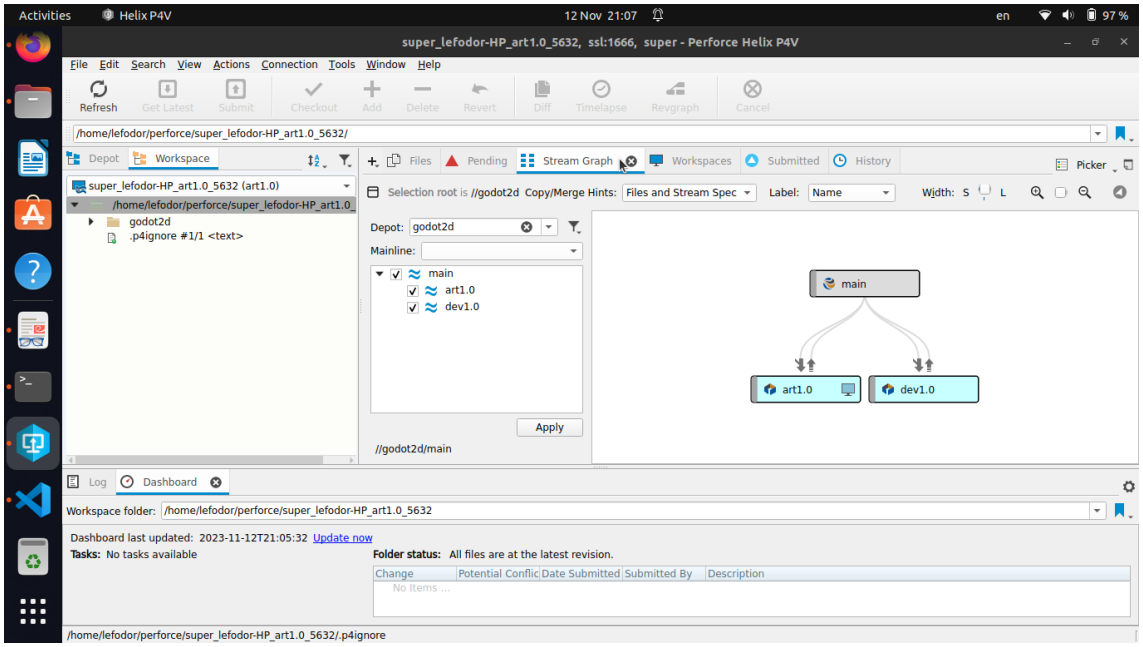

Figure 28: new art stream created

At the moment, all up- and downstream arrows between the streams are gray, indicating there are no differences that need synchronization. Let's check out the *art* directory from the *art1.0* workspace and try to make some changes to the player's sprite ($workspace/godot2d/art/playerGrey_*.png).

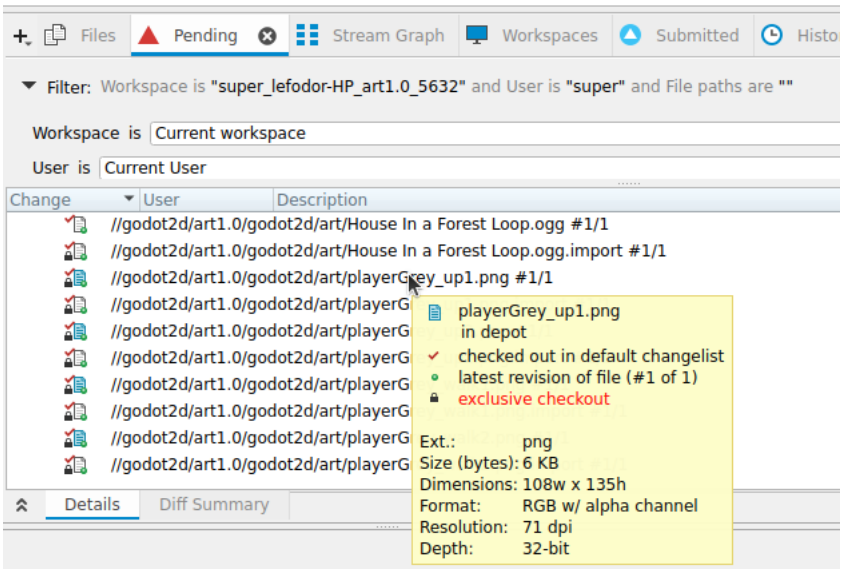

Figure 29: assets checked out

The .png files are all checked out with exclusive flag, meaning that they cannot be checked out simultaneously in multiple workspaces hence no parallel editing is possible. This property is defined in the Typemap file in 4.3. Submit the modified files and update the streams:

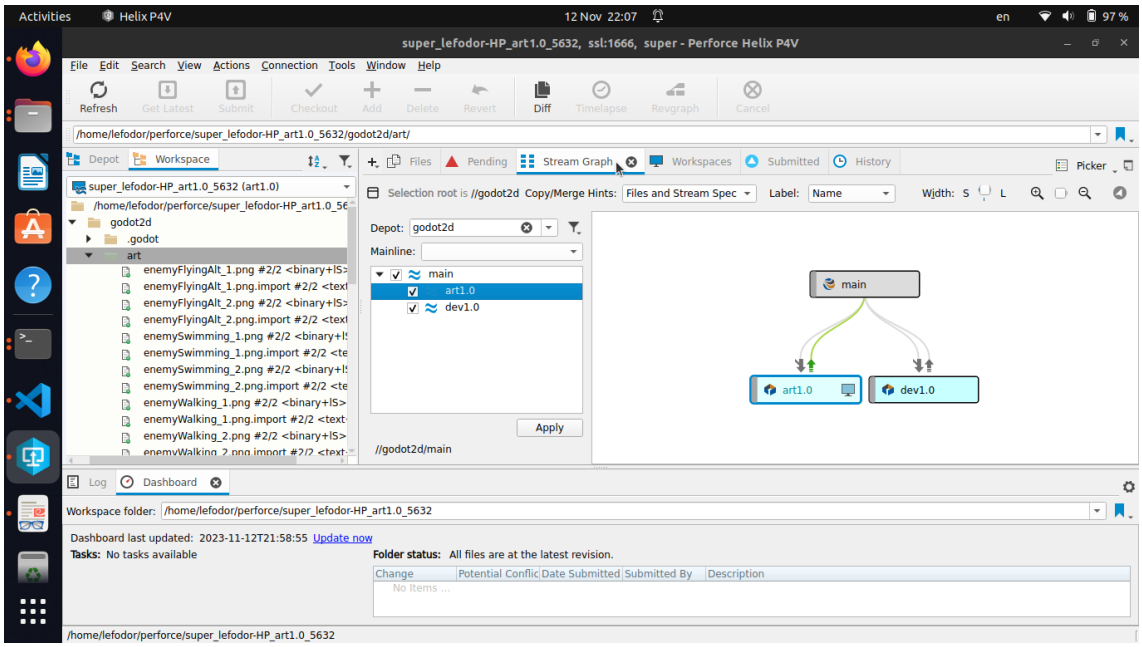

Figure 30: assets modified streams

This time, the green arrow shows up between streams *main* and *art1.0*, indicating that the latter has changed and the modified files can be copied to main. Copy the modified files from *art1.0* to *main*, and submit the changes so that they take effect.

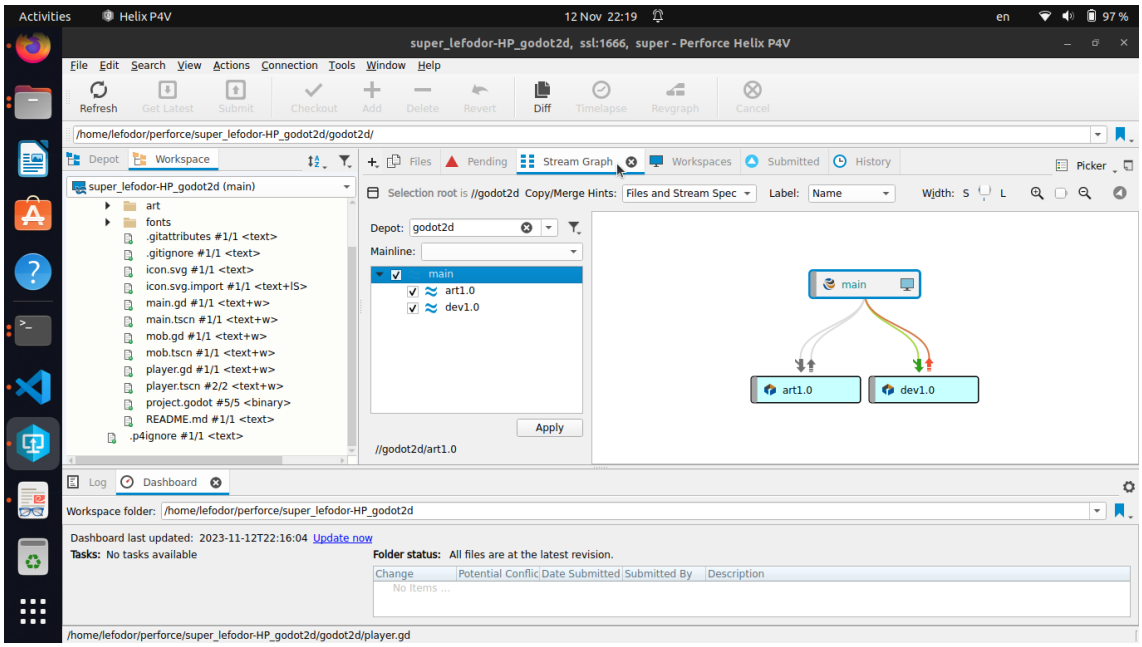

Figure 31: assets change main dev

As can be seen, the changes of from *art1.0* propagate to all other streams via *main* and require synchronization so that it gets to all working streams in the project. This can be done by following the steps set out in the previous sections.

6 Conclusion

The paper gives a brief overview about the topic of version control in game development and guides through a demonstrative case study using **Helix Core**™ from **Perforce Software** while

developing a basic 2D game with open source **Godot Engine**.
The first section (2) talks about version control in general, touch bases on main concepts and principals that show up in the industry. The following section (3) investigates some aspects of game development process and discusses differences and similarities compared to traditional software development. The third and fourth parts (4 and 5) demonstrate the version control system in action, submitting changes, creating/merging/synchronizing branches from the point of view of game designer/coder and from that of art teams using dedicated streams and propagating changes from here throughout the whole project.

References

[1] perforce.com. *The State of Game Development Report: 2020 & Beyond*. 2020. URL: https://www.perforce.com.

[2] *Gitflow workflow*. URL: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow. (accessed: 13.11.2023).

[3] *Godot Docs*. URL: https://docs.godotengine.org/en/stable/. (accessed: 20.10.2023).

[4] *Helix Core Installation*. URL: https://help.perforce.com/helix-core/quickstart/Content/quickstart/admin-install-linux.html. (accessed: 20.10.2023).

[5] *Helix Core Visual Client (P4V) Guide*. URL: https://www.perforce.com/manuals/p4v/Content/P4V/Home-p4v.html. (accessed: 20.10.2023).

[6] *How to use Perforce - Streams*. URL: https://www.perforce.com/blog/vcs/how-use-perforce-streams-101. (accessed: 13.11.2023).

[7] Aman Kumar. *GAME DEVELOPER VS SOFTWARE DEVELOPER*. URL: https://medium.com/game-developer-vs-software-developer/game-developer-vs-software-developer-f04efbf95bb2. (accessed: 15.11.2023).

[8] Giancarlo Lionetti. *What is version control: centralized vs. DVCS*. URL: https://www.atlassian.com/blog/software-teams/version-control-centralized-dvcs. (accessed: 20.10.2023).

[9] *Microsoft Team Foundation Branching*. URL: https://learn.microsoft.com/en-us/azure/devops/repos/tfvc/branching-strategies-with-tfvc?view=azure-devops. (accessed: 13.11.2023).

[10] *Setting Up Your CI/CD Pipeline: Jenkins and GitHub*. URL: https://www.blazemeter.com/blog/cicd-pipeline-jenkins-github. (accessed: 14.11.2023).