Software Studio
軟體設計與實驗

Game Development Tips

Hung-Kuo Chu
Department of Computer Science
National Tsing Hua University

CS2410

Outline

- Motivation
- Serialization problems
  - Introduction to game engine serialization and the merge conflicts they may cause.
- Merging
  - Game architecture to make merging trivial.
- Rollback
  - Using Git to save your project from peril.

Before we start…

- If you are still afraid of using Git because the command line interface scares you…
- **VS Code’s Git integrations** and **GitHub Desktop** (see Appendix) are both GUI that can simplify the process!
- Don’t give up on Git yet! You can’t run from it forever, so take this chance and start now!

Motivation

- You're a group of programmers working on the same game.
- You must work *together*, but what are the implications?
  - Communication overhead
    - Time spent on getting ideas across each other.
  - **Synchronization overhead**
    - Time spent on waiting for another person, and time spent on merging changes from different people.

Motivation

- You can reduce time spent on waiting by increasing the degree of parallelism.
  - More people working on separate modules that can be designed separately.
  - This just requires **good planning**.
- But with increased parallelism, you'll have to deal with greater synchronization overhead.

Motivation

- Students in the past years have faced serious troubles with this overhead.
- One group reported to have encountered **over 700 lines of merge conflicts**.
- We will teach you how to plan your workflow and software architecture to avoid those nasty problems!

Version Control

- In the first two weeks of this course, we introduced version control with **Git**, which helps us develop software asynchronously.
- It is important to remember that Git is designed with **human-written code** in mind.

Version Control

- But, in a game project, there are **large binaries (.png, .wav, etc.)** and **computer-generated data**, the latter of which does not work well with most VCS.
- That doesn't mean you shouldn't use Git – It's the best choice available, but there are important things to watch out when using it.

Project Structure

- The project structure of a video game differs greatly from other software.
- While they all have a lot of code in it, game projects also have complex **serialized data** which need to be read by the underlying game engine.
- These data are the primary cause of **hard-to-resolve merge conflicts** and **cryptic bugs**.

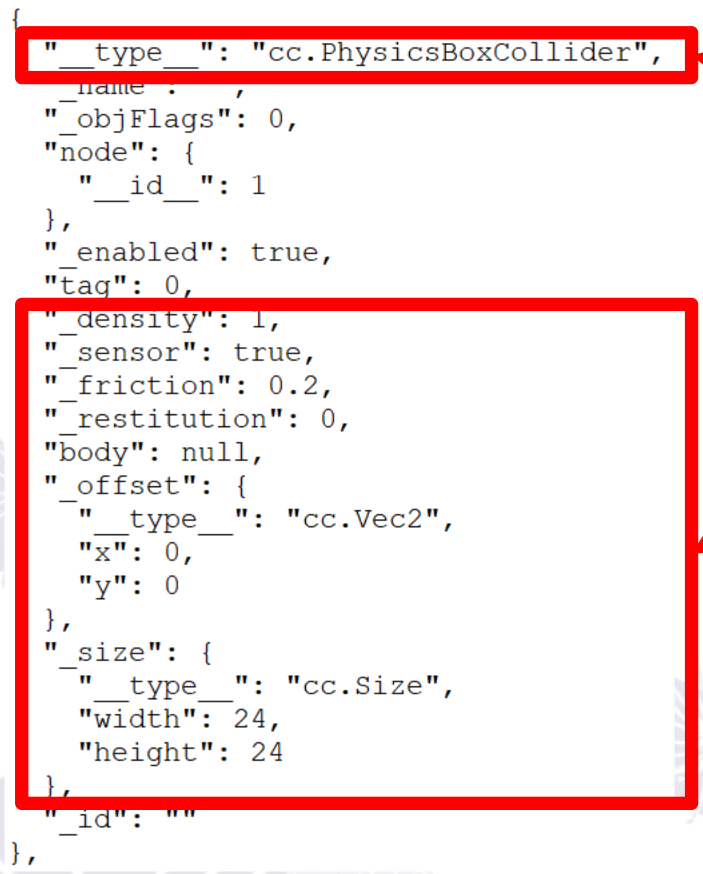
“It says my scene has merge conflicts, but I don’t know what any of the lines mean…”

SERIALIZATION PROBLEMS

Serialization

- Most game engines produce serialized metadata (.meta) for whatever **asset** you import into it.
- **Scenes** (.fire) and **prefabs** (.prefab) are also serialized data a game engine reads to create objects in the game.
- A common format used is the **JavaScript Object Notation (JSON)**, which is for the most part human-readable.

Serialization Example

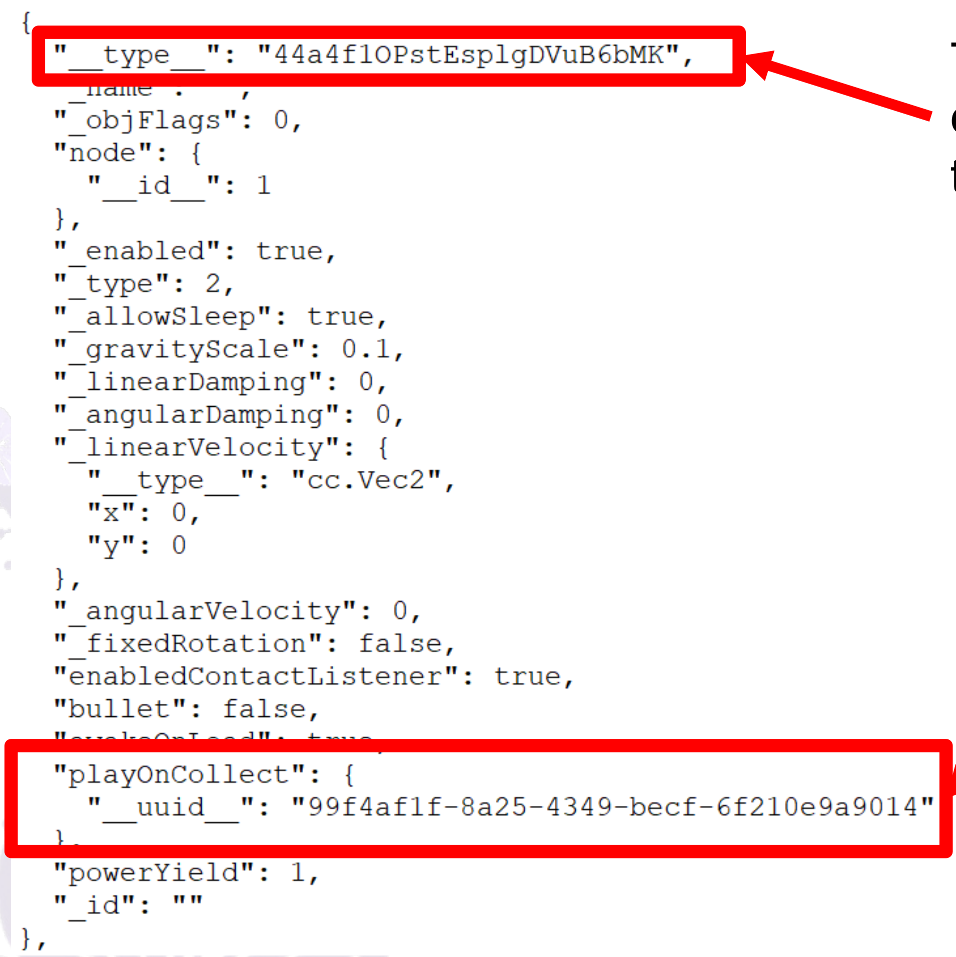


```
{
  "__type__": "cc.PhysicsBoxCollider",
  "_name": "",
  "_objFlags": 0,
  "node": {
    "__id__": 1
  },
  "_enabled": true,
  "tag": 0,
  "_density": 1,
  "_sensor": true,
  "_friction": 0.2,
  "_restitution": 0,
  "body": null,
  "_offset": {
    "__type__": "cc.Vec2",
    "x": 0,
    "y": 0
  },
  "_size": {
    "__type__": "cc.Size",
    "width": 24,
    "height": 24
  },
  "_id": ""
},
```

This JSON segment is describing a component of type “cc.PhysicsBoxCollider”

These fields below would then correspond to its properties, which you would usually edit through the editor GUI.

Serialization Example



```
{
  "__type__": "44a4f10PstEsplgDVuB6bMK",
  "_name": "",
  "_objFlags": 0,
  "node": {
    "__id__": 1
  },
  "_enabled": true,
  "_type": 2,
  "_allowSleep": true,
  "_gravityScale": 0.1,
  "_linearDamping": 0,
  "_angularDamping": 0,
  "_linearVelocity": {
    "__type__": "cc.Vec2",
    "x": 0,
    "y": 0
  },
  "_angularVelocity": 0,
  "_fixedRotation": false,
  "enabledContactListener": true,
  "bullet": false,
  "awakeOnLoad": true,
  "playOnCollect": {
    "__uuid__": "99f4af1f-8a25-4349-becf-6f210e9a9014"
  },
  "powerYield": 1,
  "_id": ""
},
```

This JSON segment is describing a component of type???

This might be referring to an audio clip, but which one?

Serialization

- The game engine recognizes assets and custom components in terms of unique identifiers, assigned when you import them into the project **through the editor**.
  - You can see these identifiers in the .meta files.
- It will then refer to them through these identifiers in a scene or a prefab’s data.
- This presents a source of **conflict**.

Serialization Conflict

- Suppose that Alice and Bob are working together on a game.
- Bob drew some sprites (.png) in his favorite painting software and used “Save As” to put them into the project. He then pushed the changes.
- Alice received the changes, opened the project in the editor, and caused the editor to assign identifiers to those sprites.

Serialization Conflict (Cont.d)

- Some time later, Bob opened the project again, and generated identifiers for his sprites. Alice told him that she had just integrated the sprites into a scene, and that she had pushed the changes.
- Bob then pulled the changes, only to be met by multiple merge conflicts over the **metadata** of the new sprites.

Serialization Conflict (Cont.d)

- What happened here is that Bob and Alice managed to generate **two different sets of identifiers** for the new sprites Bob had made.
- When Alice pushed the changes made to the scene, the scene’s metadata refers to those sprites in terms of the identifiers in **Alice’s** branch.

Serialization Conflict (Cont.d)

- In this simple case, **Bob should use Alice's version**, but things could get tricky if Bob had made other changes in the meantime, or if they had been working with a third person.

Serialization Conflict (Solution)

- To avoid this situation altogether, Bob should’ve pushed the identifiers **alongside** the new sprites.
- To ensure that this is always the case, Bob should intentionally trigger the game engine’s import process by moving the new sprites’ files into the project **through the editor**.

Class Serialization

- Another big issue with serialization comes from converting a class definition into a script component.
- Internally, custom components must be registered to / unregistered from the project's records. This is done when **the script containing the component is first imported. (Usually upon creation)**

Class Serialization (Solution)

- If you modify the component’s name or delete it outright **without the game engine’s supervision**, it might not be able to detect the changes and change the metadata accordingly, resulting in error messages that wouldn’t go away.
- **For maximum safety, when removing a component or changing its name, you should always do it through the editor.**

I merged your branch with mine and everything broke…

MERGING

Merging

- In game development, merging branches can be very problematic, precisely because of the metadata changed through the editor.
- Without proper planning, solving merge conflicts can waste **hours** of your precious time and introduce unwanted **bugs**.

Naïve Merging

- Suppose that Alice and Bob are editing the same scene at the same time. They agreed **not to change what was already in the scene**.
- Alice added some UI components, while Bob added some sprites to the scene.
- And then, Alice pulled from Bob’s branch directly, merging the changes.
- Is this enough to avoid merge conflicts? **No!**

Naïve Merging

- Now, Alice must resolve the merge conflict.
- She peeked at the scene's metadata and found **hundreds of lines in conflict**.
- She could edit it herself, manually reconstructing the merged version by rearranging the JSON objects.
- But it's **error prone** and **a waste of time**.
  - If one of them had rearranged the scene's node tree, this would be even harder to do!
- Can we do better?

Better Merging

- This time, Alice and Bob agreed to implement their changes in **separate scenes**.
- After they're both finished, Alice pulled from Bob's branch, and received Bob's scene.
- Following Bob's instructions, Alice **copied the nodes in Bob's scene and pasted them into her own scene**, completing the merge.
- **No metadata editing was needed.**

Better Merging

- Why does this work?
  - Alice and Bob's scenes are **mutually exclusive**.
  - Merging the two scenes is done manually through the editor in a third scene, **which only one person will edit at a time**.
  - Therefore, no merge conflicts!

Even Better Merging

- With good planning, we can **eliminate the need for manual merging** altogether.
- This time, Alice and Bob agreed to **put their changes under their corresponding prefabs**.
- Alice put a script in the scene, which will **instantiate** their prefabs when the scene loads.

Even Better Merging

```
const {ccclass, property} = cc._decorator;

@cclass // Place this component on a node in the scene.
export default class MainSceneManager{
  @property(cc.Prefab)
  UIPrefab: cc.Prefab = null; // Assign by drag-and-drop.
  @property(cc.Prefab)
  actorsPrefab: cc.Prefab = null; // Assign by drag-and-drop.
  // …
```

Even Better Merging

```
  onLoad(){ // Construct the full scene.
    let UI = cc.instantiate(this.UIPrefab);
    let actors = cc.instantiate(this.actorsPrefab);
    UI.parent = cc.Canvas.instance.node;
    actors.parent = cc.Canvas.instance.node;
    // PlayerController can now use cc.Canvas.instance.node.find("UI")
    // to get the node that has UIManager attached.
  }
}
```

Even Better Merging

- With this method, there is no need to edit the scene anymore. **Merging can happen without generating any conflict.**
- The only downside is that they cannot preview how the final scene looks. They have to enter play mode to see the combined results.

Even Better Merging

- Is not being able to see the final scene in the editor that big of an issue?
  - The editor is most useful for adjusting **visual** parameters, like position, scale, rotation, etc.
  - These parameters can be edited **inside the prefabs**, rather than in the scene view.

Dependencies

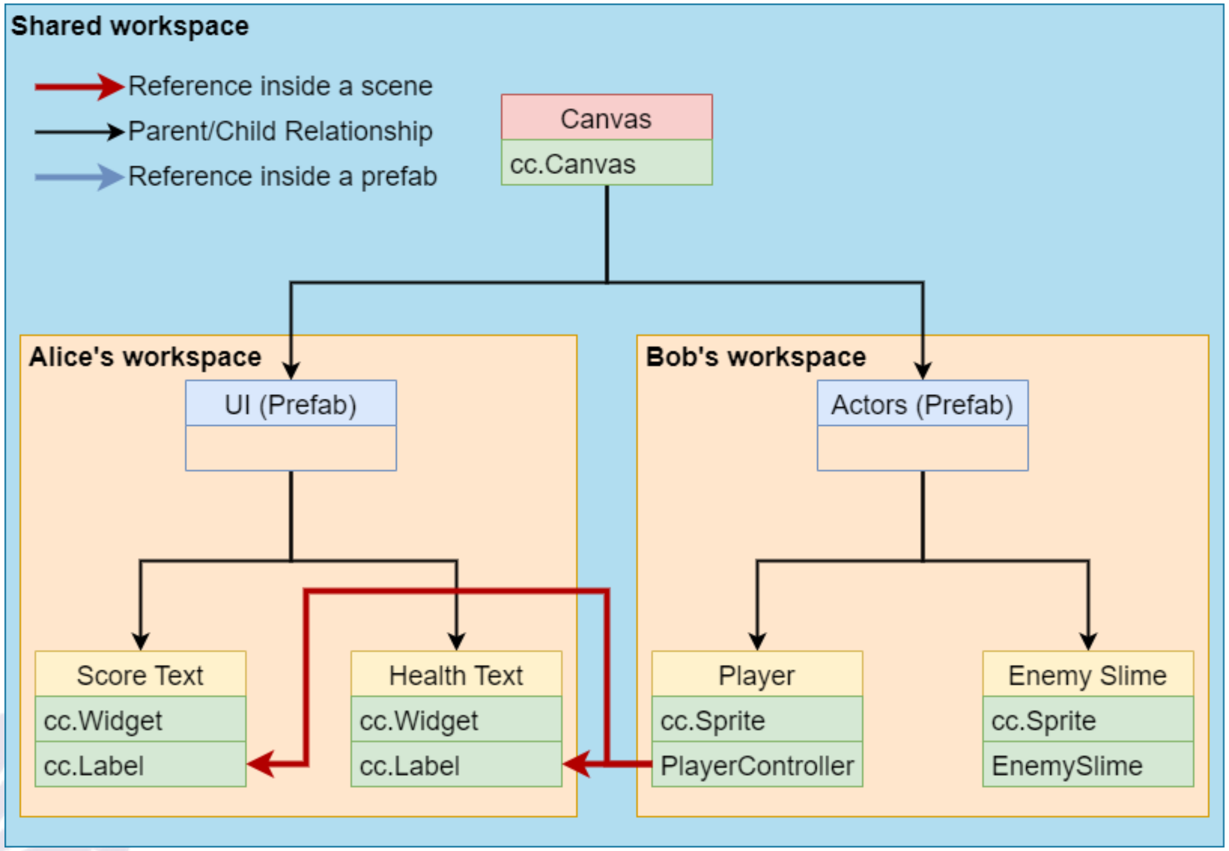
- It is important to note that the task of merging Bob’s scene into Alice’s scene is trivial because they have no **mutual dependency** yet. Bob does not need anything from Alice’s scene, and vice versa.
- Furthermore, they both don’t need anything from the original scene.
- In software engineering, a dependency means that one object **references** another.

Dependencies

- Identifying dependencies when planning your project can help you break a project into smaller chunks of work **which can be worked on separately**.
- However, most objects have at least a few dependencies.
- In the case of Alice and Bob, Bob might need to reference Alice’s UI scripts, telling them the player’s current HP and lives.

Dependencies

- The scene hierarchy would look like this:



- Bob needs to enter Alice’s workspace!

Dependencies

- It seems that Bob needs to assign the reference to Alice’s UI using their nodes’ paths. (with `cc.find`)
- This would **violate encapsulation** between the two modules – **Bob needs to know about the structure of Alice’s prefab.**
  - If Alice renamed any node on the path or changed her prefab’s node tree, Bob’s scripts would suddenly start failing to find the references.

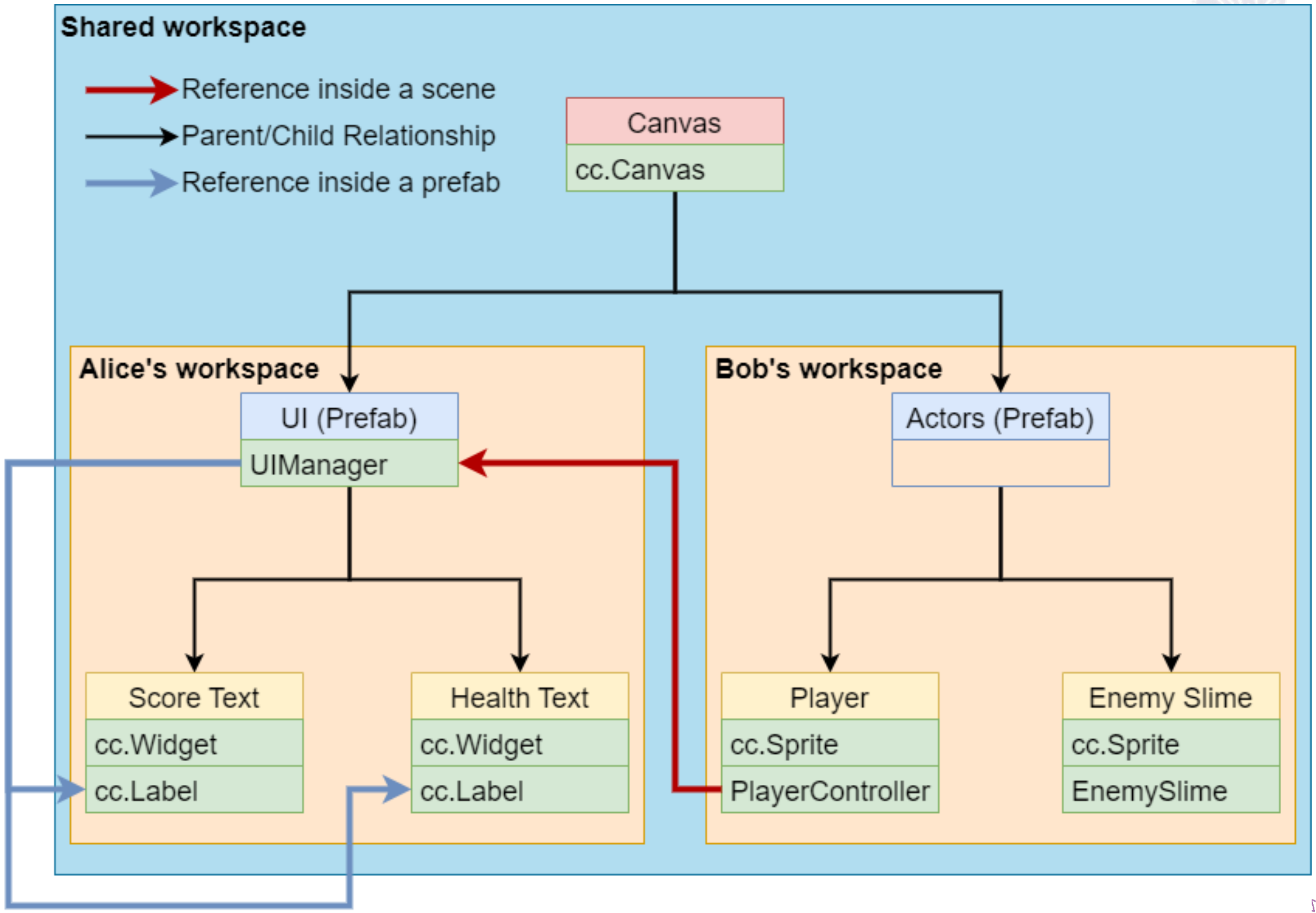
Abstraction

- That Bob needs to inspect Alice’s prefab indicates that there isn’t enough **abstraction** between the two prefabs.
- An **abstraction layer** can hide implementation details with **indirection**, becoming a “middleman” across multiple objects.

Abstraction

- To maintain encapsulation, Alice could collect all references Bob might be interested in into one component, preferably placed in the prefab's root.
- Bob could then retrieve the references from that component, rather than exploring Alice's prefab's node tree.
- We say that the component **abstracts the exact locations of the references.**

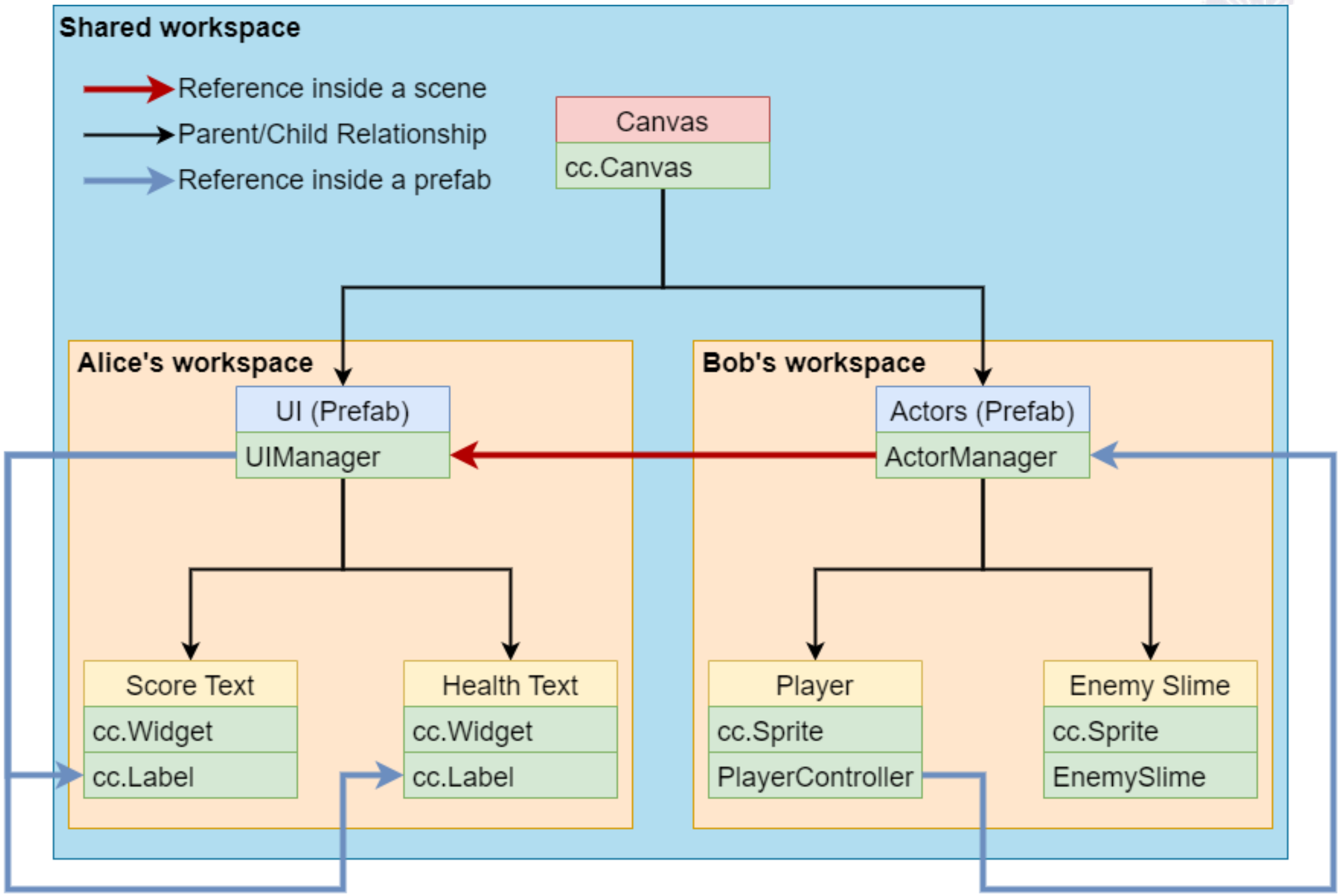
Abstraction



Abstraction

- Now Alice just needs to maintain the UIManager's references on her own, and Bob does not need to know about the exact locations of the Score and Health Texts.
- However, Bob still needs to reference UIManager by the instance of Alice's prefab in the scene. (Again, with `cc.find`)
- With **one more layer of abstraction**, Bob can **avoid checking the scene's node tree** altogether!

Abstraction



Abstraction

```
// In MainSceneManager:
onLoad(){ // Construct the full scene and resolve references on load.
  let UI = cc.instantiate(this.UIPrefab);
  let actors = cc.instantiate(this.actorsPrefab);
  UI.parent = cc.Canvas.instance.node;
  actors.parent = cc.Canvas.instance.node;
  actors.getComponent(ActorManager).injectDependencies(
    UI.getComponent(UIManager)
  ); // Give ActorManager the reference to UIManager
  // Now PlayerController can reference UIManager through
  // ActorManager, all inside the Actors prefab.
}
```

Abstraction

- With this model, Alice and Bob could connect ActorManager and UIManager right when their prefabs are instantiated.
- Even if the actual objects they're referencing don't exist yet, **these intermediate references can be connected beforehand!**
- This could even be the job of **a third person**, who doesn't need to know the exact structure of the two prefabs' hierarchies.

Abstraction

- It is true that with abstraction layers in the way, PlayerController needs **up to three references** to access the cc.Label components (as opposed to just one without), possibly causing a performance penalty.
- This can be mitigated by **caching the references** inside PlayerController.

Divide and Conquer

- It is now apparent that organizing scenes in terms of prefabs can greatly **reduce the time spent on merging** and **accelerate development speed**.
- We can take this approach to **break down prefabs into smaller prefabs as needed**, so that more people can work without waiting for each other.

Solving the problem by pretending it didn’t happen.

ROLLBACK

Rollback

- It is not uncommon for your game project to become beyond saving.
  - When your game engine **crashes**, some metadata may be **corrupted**.
  - There might be too many merge conflicts or bugs for you to solve after merging.
- It's important to realize that sometimes it's faster and easier to rollback to an older version.

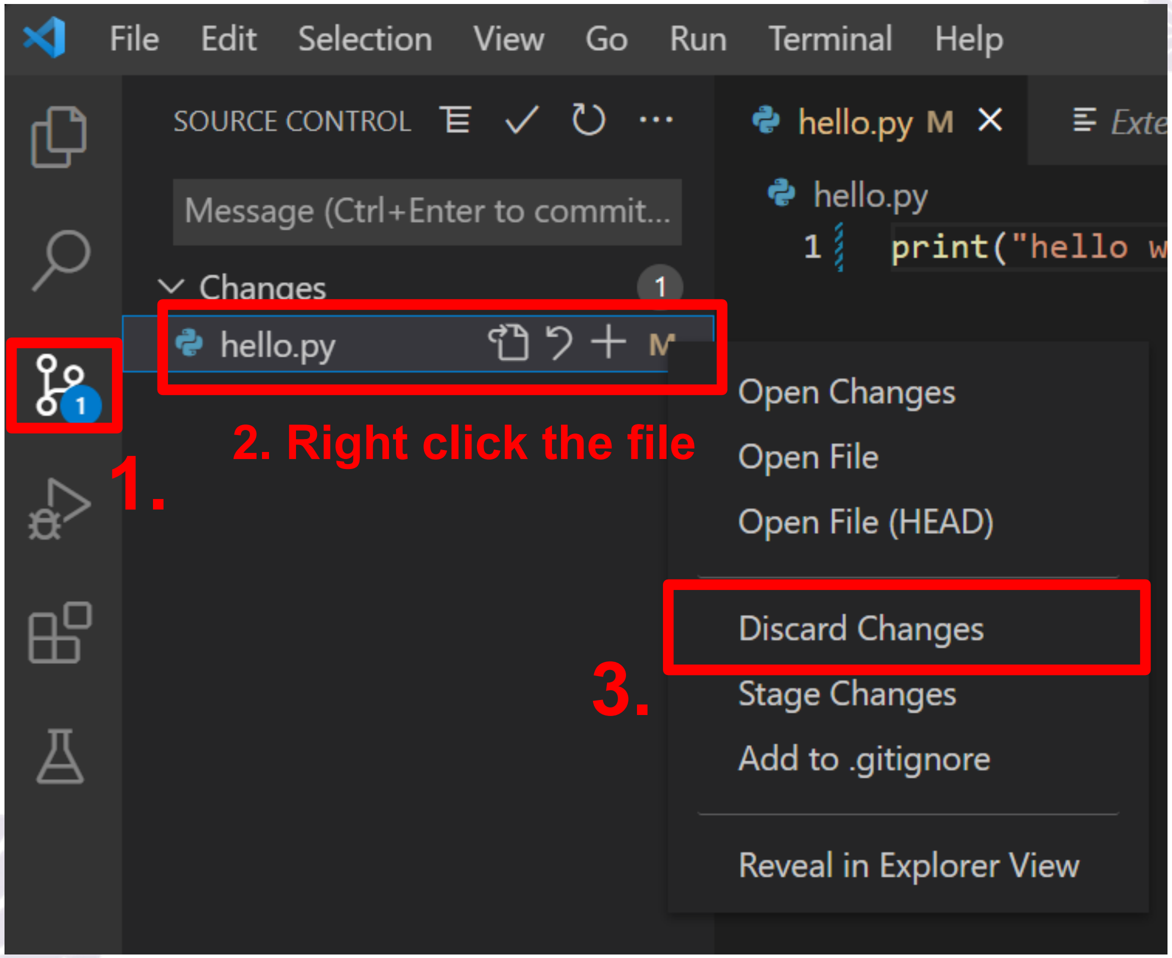
Discarding changes

- If it’s just that your game engine crashed and corrupted some data, you can discard the files that were changed unexpectedly.
- For git, you can use **git checkout** to discard uncommitted changes to a file.

```
git checkout -- <filename>
```

Git – git-checkout Documentation

Discarding changes (VS Code)

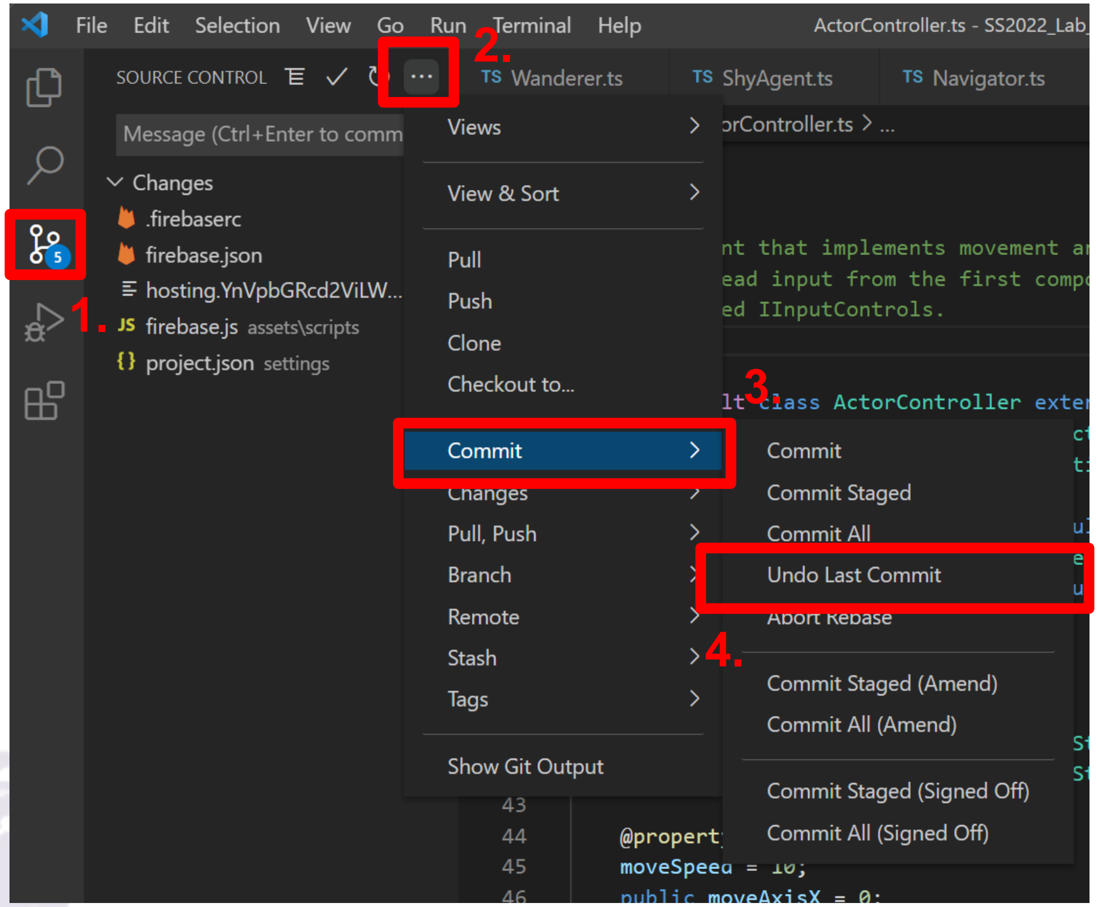


```
print("hello w
```

Undoing commits

- Usually, when something goes wrong, it’s because of a recent commit, such as the commit that comes from merging.
- In VS Code, you can use the **“undo last commit”** button to put all changes back to the list of uncommitted changes, which you can then discard or keep.

Undoing commits



Reverting commits

- You can also use the more powerful **git revert** command, which can revert any commit, and not just the last one.
- **This will create a commit afterwards.**
- For VS Code, you'll need extensions such as **Git History** or **GitLens**.
- See the documentation for a detailed explanation.

Git – git-revert Documentation

Closing Notes

- While this course is about making (frontend) software, it is not a course about **software engineering**, and so these slides only cover the bare minimum for **small-scale development**.
- Consider taking the course Software Engineering (CS446100) to learn more about this topic.

Takeaways

- Please use Git.
- Only ever add files to the project **through the editor GUI**.
- Avoid having more than **one person** modifying the same scene.
- Break down a scene into independent **prefabs** and use **abstraction layers** to make merging trivial.

thank you!
Question?