Enhancement Proposal

Super Mario ... Oops I mean Tux!

Gabriele Cimolino, Jack East, Meryl Gamboa, Tyler Searl, Matt Skoulikas, Stefan Urosevic

2017-12-1

Contents

List of Figures

Abstract

In assignment 2 we proposed the addition of a shop which would sell cosmetic items and gameplay bonuses to the player. We went back to our conceptual architecture for SuperTux and determined two possible implementation strategies for this feature. The two were compared, by means of a SAAM analysis, in terms of their non-functional requirements and the impact that each approach would have on the interests of the game's stakeholders. Having decided on the Persistent Shop, we implemented the feature and explored its implementation and interactions with the rest of the game using the techniques which were used to create our revised conceptual architecture and better understand how the game works. Finally, our impressions of the process are explored.

Enhancement Proposal

The feature which we proposed in assignment 2 is a shop where the player can purchase new sprites, called cosmetics, new sounds, and extra abilities, called Shop Moves, as well as the normal bonuses available in the game by picking up the flowers placed throughout SuperTux's levels. We've considered two possible implementations of this feature and chosen one which we've implemented. This report will detail the process that we followed in its creation and give our impressions of that process itself.

Motivation for Feature

The motivation for this feature was to give new purpose to the coins collected while playing the game. In its current form Tux has no use for coins other than as a currency with which the player pays for respawns when Tux dies, and as a metric for the player's completion of any level. These features certainly justify the existence of coins in the game; however, they lack any incentive for new players to engage with the mechanic. Our hope is that the Shop system will provide the desired incentive by creating a way to reward the player for their diligence in collecting this currency.

Cheat Menu

Currently, the feature which most closely resembles what we had in mind is the cheat menu, which offers bonuses to the player from a menu in the world map. This feature, however, is only accessible if the game is run in debug-mode, which needs to be specified at runtime by setting the debug-mode flag. This feature, although similar to what we wanted, is difficult to use and potentially game breaking since it can be used without restriction, making the game much easier. For these reasons we believe that the Shop system would be an improvement to the game that would make the game more enjoyable without debasing the designers' vision for how the game should be played.

Possible Approaches

After some time spent with the SuperTux source files, we drafted two possible implementations of the Shop.

Temporary Shop

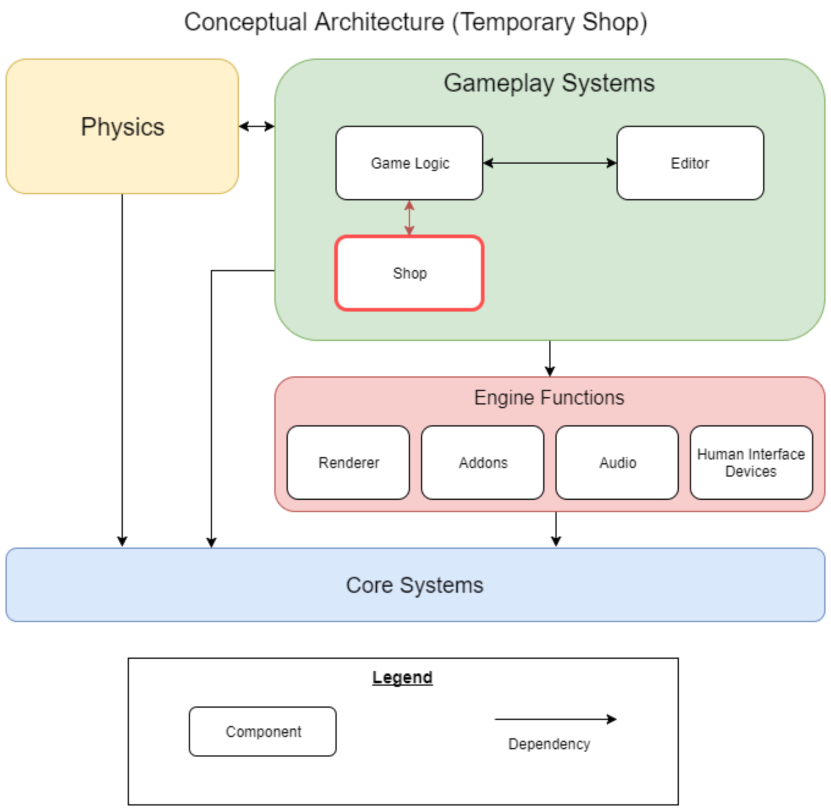

Figure 1: Conceptual (impact) architecture diagram of Temporary Shop implementation with new systems and dependencies marked in red

The implementation which we considered first is one which we've called the Temporary Shop because it would function much like the cheat menu in that it would allow the user to buy Shop Items and use them for the rest of their game session. Once the player exits the game, the Shop Items which they have purchased are no longer available upon returning to the game and must be purchased again in order to continue being used. Since this implementation of the Shop would reinitialize itself at runtime, with no long term retention of purchases, it could be implemented as a subsystem of Gameplay Systems.

This implementation would therefore only require modifications to Engine Functions and Gameplay Systems in order to add a new input for the currently selected Shop Move, add the Shop Menu to the world map menu, and to make the necessary modifications to the already present systems which would make use of the Shop. In terms of its dependencies, since this implementation allows the Shop to be a subsystem of Gameplay Systems, only a single codependency between Game Logic and the Shop would be required. The Shop would depend on Game Logic for information about the current state of the game, such as the number of coins the player has, and Game Logic would depend on the Shop to make selections from the Shop Menu and to get information about which Shop Items have been purchased and which are selected for use. The Shop needs no, and often cannot have, interactions with Editor because the Shop is only available from SuperTux's Story Mode, which cannot be used concurrently with the Editor, and requires none of the other functionality that Editor provides, such as the functions that extract information from objects that inherit from GameObject.

The Shop's implementation makes use of two design patterns; these being the singleton and template patterns. As well, our intention was to make use of the facade pattern when implementing the Shop, such that the Shop object would be the system's facade and the Shop Menu would be another way of interacting with that facade. However, because of how SuperTux's menu system works and the coding conventions which were established in all of the other menus' implementations, this plan was scrapped in order to make the Shop Menu's implementation more consistent with the rest of the menu system and to avoid creating unnecessary dependencies in the Shop.

A singleton pattern is used by the Shop object itself. The singleton pattern ensures that only a single instance of a class exists at any given time[2]. SuperTux already makes use of this pattern in its abstract Currenton class, the class from which all of the system management objects like SoundManager and Editor inherit, which acts as a global variable storing an instance of the concrete child class. This allows any object that depends on Shop to call its current function to get a reference to the current instance of the Shop.

The template pattern is used by the FlipLevelTransformer object, which the Shop needs to be able to perform the Flip Move. The FlipLevelTransformer inherits from the abstract class LevelTransformer, a template for two functions called transform and transformSector. These functions are used to transform a SuperTux level during play and FlipLevelTransformer's implementation specifies that the level should be inverted when its transformSector function is applied to it. In this way, similar level transformation objects could be created and used as Shop Moves to give the player other interesting new gameplay options that allow for complex strategies involving interactions between Shop Moves and other game mechanics.

Persistent Shop

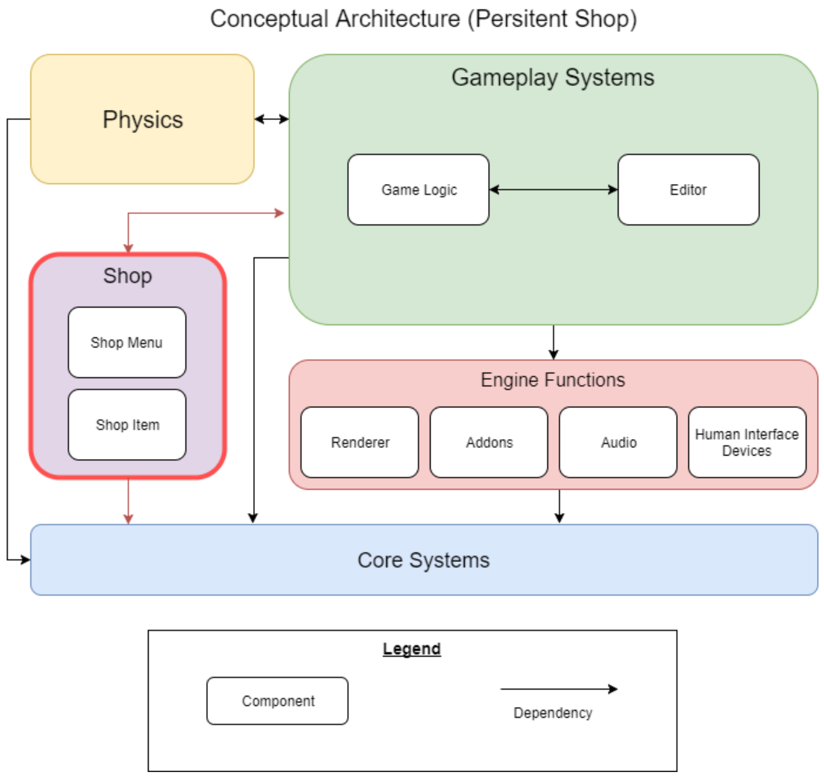

Figure 2: Conceptual architecture diagram with Persistent Shop implementation

The Persistent Shop approach is a variant on the Temporary Shop, allowing the game to save information about the Shop to be reloaded during initialization. Initially we did not believe that we would be able to implement this feature because we believed that it would require the development of a new type of save file specifically for the Shop. The way in which SuperTux saves the player's progress is a complex system involving abstracted file system functionality on top of PhysFS, the external resource management system it employs. Adapting this functionality for a new purpose, without much understanding of how the system currently handles this problem, did not seem possible within the time constraint. This was the reason for the distinction between approaches since we believed that one way was feasible while the other wasn't. However, after fully implementing the Temporary Shop we realized that the creation of a new save type was not necessary if we made modification to the current game save process to include Shop information in the player's save file.

This implementation of the Shop would have the same codependency with Game Logic as the Temporary Shop, visualized in the diagram as a codependency with Game Logic's parent component Gameplay Systems, as well as a new dependency on Core Systems to grant access to the objects involved in the saving and loading process. When the game is saved, the Shop is passed a Writer object which it uses to add information about the Shop to a predetermined section of the player's save file. Similarly, when the player's save file is loaded the current instance of the Shop is passed a ReaderMapping object which it uses to extract Shop information in much the same way. This is the approach that SuperTux already takes for saving information about the Player Status object and about each of the game's levels.

SAAM Analysis

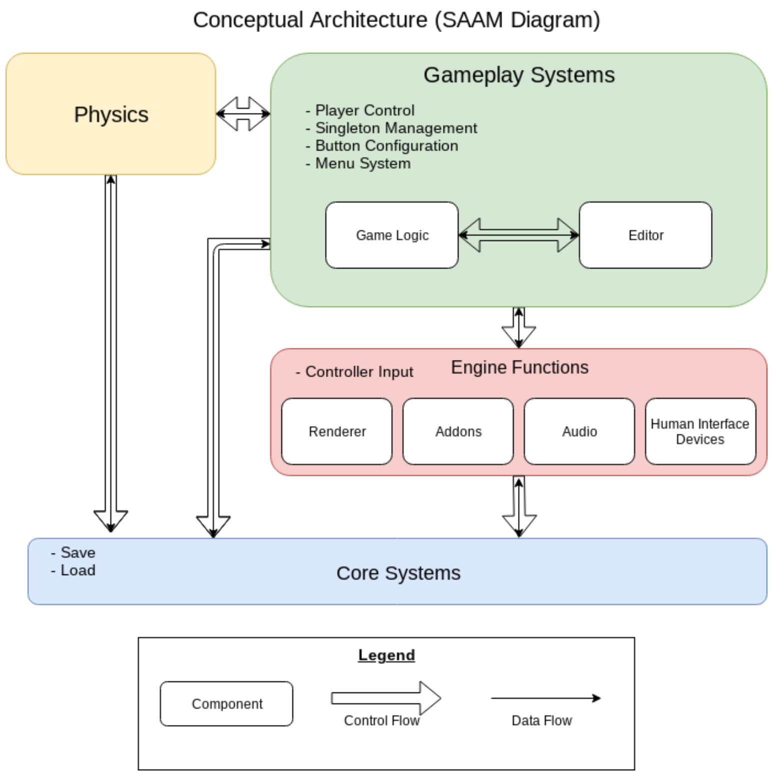

Figure 3: Diagram of our conceptual architecture with the data flow and control flow of dependencies shown and functionality required to implement the Shop grouped in its containing subsystems.

Once we had defined the two approaches that we believed were most sensible given the established architectural and coding style, we performed a SAAM analysis by comparing the changes that would needed to implement the Shop in these two ways and the effects that these changes would have on the system in terms of its non-functional requirements and stakeholder interests.

Interactions with Subsystems

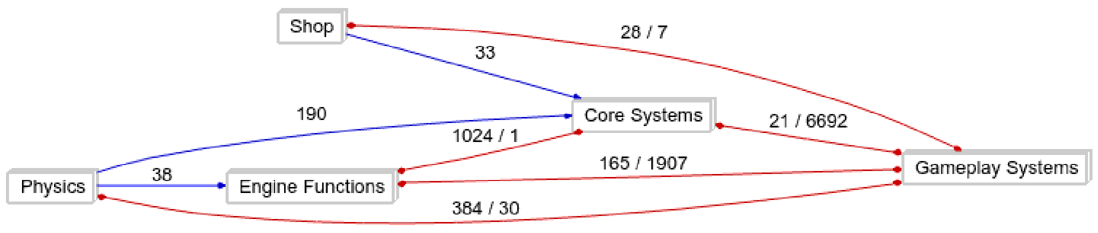

Figure 4: The Understand dependency graph generated when the Shop source files are added to our Understand architecture.

When this feature was proposed, we believed that this sort of architectural change would require changes to and coupling with potentially every subsystem in our conceptual architecture. However, upon inspecting where these changes would be required and determining what sorts of changes were feasible and consistent in style with the way that the rest of the game is implemented, we realized that almost all of these modification would be most appropriately made to the systems in the Gameplay Systems subsystem. This means that, instead of being as highly coupled as we anticipated, we were able to implement the Shop with the modification of only two subsystems, Gameplay Systems and Engine Functions, and coupling with only Gameplay Systems and Core Systems. The ability to create such a cohesive and sparingly coupled Shop subsystem allowed us to implement these features without changing the system's architecture, since it could be implemented in much the same way as Physics is, while having minimal effects on SuperTux's quality requirements.

Effects on Architecture

The Shop, even with the ability to save and load, only requires two dependencies which are extremely common for similar systems to have. Much of the functionality found in Gameplay Systems have similar dependencies on the subcomponents of Gameplay Systems, such as the menu system or the world map, and functionality in Core Systems, such as loading resources. Adding such a feature would not require changes to the system's architecture at all and would be indistinguishable from many other systems in terms of its dependencies.

Effects on Non-Functional Requirements & Stakeholders

The only stakeholders whose interests would be affected by the introduction of the Shop into SuperTux are the developers and the players. Editors would only be indirectly affected, by players who use Shop Items in their levels, and so the Shop's effects on their interests cannot be considered with much certainty. These are some of SuperTux's quality requirements and the Shop's potential impact on them.

Performance

There is no significant performance difference with the Persistent Shop or the Temporary Shop. In fact, both implementations have nearly no effect on performance.

Evolvability

The Persistent Shop implementation requires a whole new subsystem to make it work properly. This is beneficial because it means that all the resources the Shop needs are merged into its own component. However, the Persistent Shop requires changing the save file format to enable cosmetics, bonuses, and moves to stay on Tux even after a gameplay session is ended. Adding a new component also increases coupling of the system and so changes to either of the systems on which the Shop depends might break it, reducing the system's evolvability. This places a constraint on the developers when considering changes to those systems in the future; however, the Shop would not be the only system to use that functionality in that way so this constraint already existed.

The implementation of the Temporary Shop involves creating a subsystem inside the gameplay systems component, which increases the cohesion of the system. Changes to subsystems other than Gameplay Systems should have no effect on the operations of the Shop and so the evolvability of the system remains unaffected.

Maintainability

The only maintenance required for the Persistent Shop implementation is adding new items to the store. If, in the future, a new item should be added then the item needs to be created in the code and the Shop would then need a new menu option to buy that new item. However, if maintenance needs to be done to Core Systems, the system in charge of various libraries, some Shop functionality would need to be rewritten to account for these changes. This maintenance, however, would be no different than the maintenance performed on the other affected subsystems and should therefore require no extra effort from the developers.

The Temporary Shop would not need to be maintained since it would have no dependencies outside Gameplay Systems and the dependencies which it would have are so commonly used that it certainly would not be the only system to be affected if changes were required. If a change were to be made to one of these systems then maintaining the Shop would be no different than maintaining any other affected subsystems, since it makes use of its dependencies in standard ways, and so the maintainability of game remains unaffected.

Modifiability

Since the Shop was built to use some pre-existing items, it was built in such a way that encourages the addition of new items. It is very easy to add items to the Shop. All someone needs to do is create the item in the code and then add an appropriate entry in the Shop menu to select that item. On the other hand, this implementation took longer than other approaches we looked at. One of the reasons for this is that it required the alteration of the save file format. The new save file format was used to enable persistent items to to work properly.

The Temporary Shop requires no new save file format and so development is much faster and cheaper. This, however, means that to enable saving in this implementation, a rewrite of the Shop, or major parts of the game, would be in order.

Testability

Because the implementation of the Persistent Shop can be done with such low coupling, testing would be simple because of the few, highly standard, uses of functionality located outside the Shop. The only functionality which is required by the Shop from outside systems is the ability to save and load information about the Shop and to gather information about the current game state. All of these uses can be found already in systems other than the Shop and so it is known beforehand what correct operation should look like. The Shop's uses of these other systems can therefore be verified by checking its results against the results of other systems.

Testing the operations of the Temporary Shop implementation would be even simpler, mostly because its only dependency would be on Game Logic which would make much heavier use of it than it would of Game Logic. The functionality that the Shop requires of Game Logic is the ability to get and set Player Status values. This means that its requirements are simple and examples of this functionality being correctly used are already common in the sources. The functionality that Game Logic requires of the Shop is not as simple in comparison but it is easily testable from within the Shop since its operations are almost entirely focused on managing its collection of Shop Items, where any errors would be made apparent by invalid or incorrect Shop Item values.

The ease with which the Shop's functionality can be tested ensures that bugs are likely to be noticed and fixed before the player has a chance to encounter them. The Shop's testability therefore makes it less likely that the player's experience of the game will be affected by bugs introduced by the Shop, negatively affecting the player's interests.

Testing

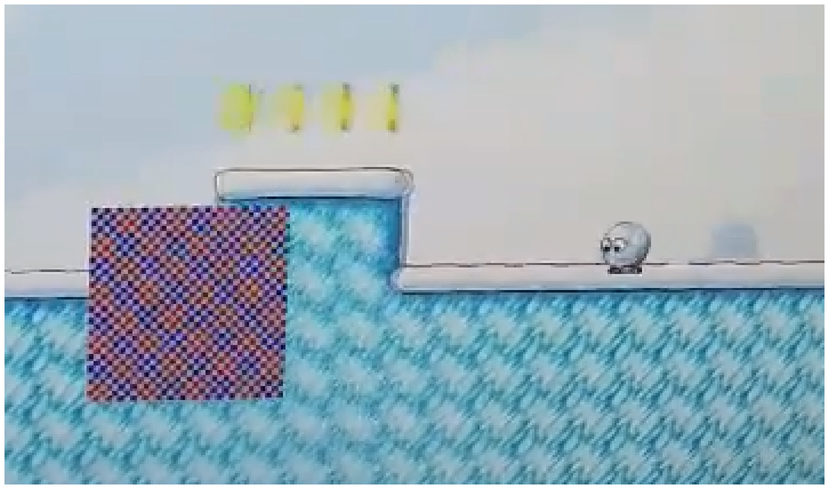

Figure 5: An real example of unexpected behaviour of a Shop resource encountered during testing

When implementing a shop feature in a game like this, testing everything properly is very important. The most important thing to test is how the Shop interacts with the rest of the game and making sure that the Shop does not remove any pre-existing functionality. Since we reference some code that already exists in the program in our Shop, testing this functionality of the Shop is less important because it is unlikely to be where potential bugs would be introduced. To make sure all the interactions work correctly, we would use integration testing. Some examples about what we would test include, changing Tux's sprite in the store, changing the sounds Tux uses, whether the player can activate the move they bought from the Shop, and whether the save game functionality still works after the Shop is used. These are all examples of how the Shop would interact with the rest of the game. Some examples of the Shop functionality being tested include how the move actually works, the functionality within the Shop, and whether the in game bonuses still work. These are things that are not too important to test because they are things that either already exist in the code, and so we can assume they already work, or they only matter within the Shop so if they don't work, they won't break the rest of the game. This is not to say these examples should not be tested; just that testing these examples is not as important as testing the interactions between the Shop and the game.

More of this sort of testing would be required again if development of the Shop were to continue to the point that this feature were actually pulled into the game. Once the basic operations of the Shop have been verified, its long term operations would need to be considered. These tests might include observing the Shop as the player plays more of the game and completes it or as modifications are made to the player's save file, like starting a new game. The Shop's behaviour under these conditions would need to be explored and defined later in testing. The best way to perform this sort of examination would be during actual play testing.

Chosen Implementation

Due to the fact that it was now possible to implement the Persistent Shop within the provided time we chose this approach and implemented it.

Use Cases

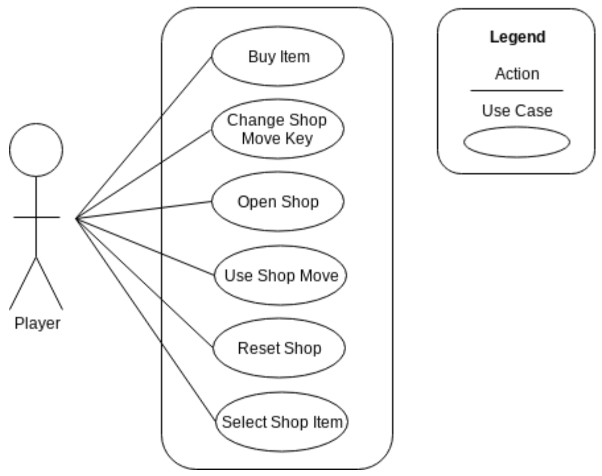

Figure 6: Use case diagram of the player's possible intentions while using the Persistent Shop

Unlike the use case diagram for the SuperTux game from assignment 1, which featured an editor primary actor, the Shop use case diagram only has one actor named player because the Shop is only accessible in Story Mode. The player actor's possible intentions when interacting with the Shop are as follows.

- Buying an item is done through the Shop Menu which is accessible through the world map. Purchases can be made by selecting the desired menu option while in possession of a sufficient supply of coins.
- The button used for the player's currently selected Shop Move can be set in the main menu by setting the Shop Move in either the keyboard configuration menu or the joystick configuration menu.
- The Shop can be opened by pressing the escape key while in Story Mode and at the world map and then selecting Shop from the menu.

- During normal gameplay, the player can press their configured Shop Move button in order use the Shop Move they currently have selected.
- If the player wishes, the Shop can be reset such that no items have been purchased. This option is available from the Shop Menu.
- Finally, the player can select which Shop Item they would like to use during gameplay, if multiple items of that type are purchasable. The currently selected item can be deselected if it is selected. Selection and deselection of Shop Items can be done using the Shop Menu.

Sequence Diagrams

Sequence Diagram: Saving the Shop

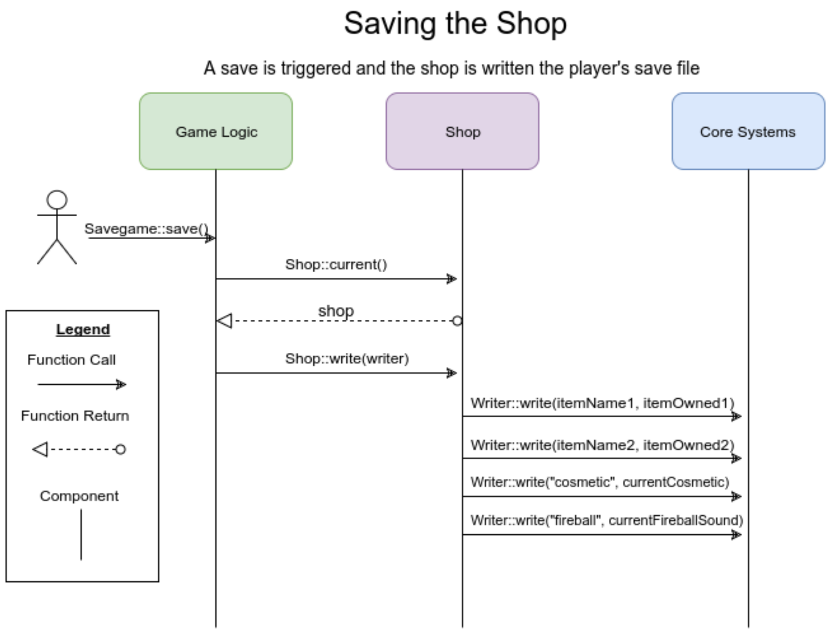
Saving the Shop
A save is triggered and the shop is written the player's save file
Game Logic
Shop
Core Systems
Savegame::save()
Shop::current()
shop
Shop::write(writer)
Writer::write(itemName1, itemOwned1)
Writer::write(itemName2, itemOwned2)
Writer::write("cosmetic", currentCosmetic)
Writer::write("fireball", currentFireballSound)
Legend
Function Call
Function Return
Component

Figure 7: Sequence diagram of the game saving the shop during its saving process.

When the Savegame object in the Game Logic subsystem receives a request to save the game it first writes the information related to the player, such as the bonus that they currently hold and the number of coins they have, and the game state, including information about level clear times and discovered secret areas. Only once this has been done does the Savegame send a request to the Shop Currenton in order to get a reference to the current Shop. After getting this reference it passes a Writer object to the current Shop so that it can use it to write information about the state of the Shop to the save file. The Shop does this by requesting that the Writer perform a write operation on two arguments, a string, either the item's name or the name of a Shop variable, and a value associated with the string, be it another string, a boolean, or a numeric value. In the case of ShopItems, the item's name and a boolean corresponding to whether it is owned are stored. In the case of a Shop variable, such as the currently equipped cosmetic, the name of the field is stored along with the field's value, possibly an enumerated identifier or a file path. Once all of the Shop's information has been written the Savegame is free to finish writing the save file and conclude saving.

Sequence Diagram: Buying a New Item

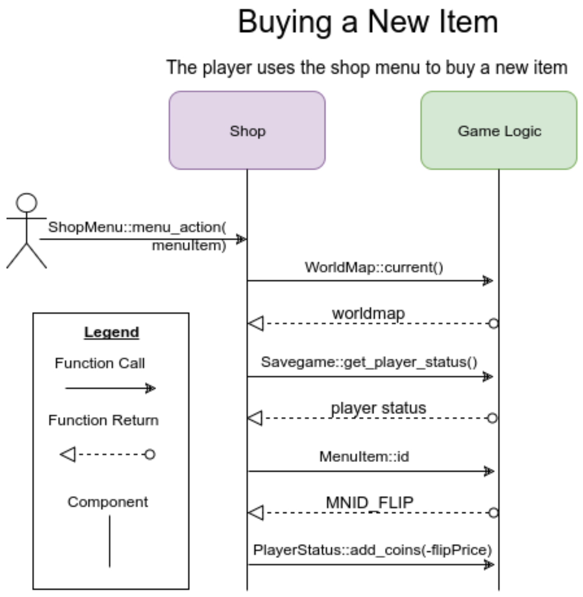
Buying a New Item
The player uses the shop menu to buy a new item
Shop
Game Logic
ShopMenu::menu_action(menuItem)
WorldMap::current()
worldmap
Savegame::get_player_status()
player status
MenuItem::id
MNID_FLIP
PlayerStatus::add_coins(-flipPrice)
Legend
Function Call
Function Return
Component

Figure 8: Sequence diagram of the player using the Shop Menu to buy a new item.

Upon receiving a menu selection from the menu system, the ShopMenu needs to collect several pieces of information about the context of the request. It first requests a reference to the current Worldmap object from the Worldmap Currenton. Using this new reference, it is able to get the currently loaded save file in order to access the PlayerStatus object. However, before it can make use of this PlayerStatus it needs to discern which menu item has been selected. It does this by getting the ID attribute from the MenuItem object and uses it as the argument of a switch to select the correct menu option. Once the case has been selected the desired item is purchased from the Shop, the price of the item is subtracted from the number of coins in the Savegame's PlayerStatus object.

Risks

Because our chosen implementation of the Shop required modifications of how the game saves the player's progress, these changes could introduce errors into the pre-existing save process which could have unforeseen side effects on the player's save file. So far, this has not been the case and we do not anticipate that any changes to how saving the game is handled in SuperTux could cause the player's save data to be corrupted by the Shop; however, we do not know all of the details of the implementation of the save feature, nor do we know all of the details of the operations of its dependencies, and we could not possibly know what future changes might be made to this system. In any event, we believe that the risk of adding this feature to the game, in its current form, is minimal if not risk free.

Implementation Process & Reflection

Concurrency

Our implementation of the Shop did not require the concurrent execution of any processes. In fact, most of the function calls to the pre-existing subsystems are required to return in order for execution of the Shop's functions to finish. There are, of course, items in the Shop that have effects on processes that require some concurrent execution, such as playback of the sound asset Shop Items during gameplay, but the Shop does not reference these function directly. At all times the Shop is either initializing itself or getting and setting variable values, all of which have no greater than linear complexity and can therefore be expected to return quickly and without the use of other threads.

Team Issues

One possible issue that the development team might have with our Shop is how to handle it in the future. Since the development team did not create the Shop, they would not be familiar with how it functions. This means that they would not be familiar with how their code is used to create the Shop. If changes or updates to the game change the functionality of some code that is used in the Shop, it could change the functionality of the Shop in ways that were not expected or even render it useless.

Another possible issue would be with the gameplay. Something developers of a game need to keep in mind is how the game is balanced. That is, making sure the game is not too hard and not too easy. Introducing a shop as a way to get powerups would be an easy way to ruin the balance of the game and is something that needs to be carefully changed to be completely fair for the player. This is also the case with the new moves we are introducing. These moves were something the developers never planned for the player to use in gameplay and were more likely to be used in alterations made to a level. Giving the player the ability to use these moves would make levels much easier to beat. An example of this can be seen in using the FlipLevelTransformer. This move can be repeatedly used in quick succession to let the player effectively fly across the level. Some solutions to this are making the price for moves and powerups expensive enough that buying them is not trivial, or changing the functionality of the moves in a way that prevents the player from abusing them.

Another issue that we did not really expect was the lack of tested and known code. Since the original superTux developers abandoned the project for a while[1], and the current development team did not work on the original project, a lot of the old and outdated code is unknown or untested. This could create issues when working around our Shop due to not being able to foresee how these unknown pieces of code will interact with our Shop, especially if they are updated to newer code.

Limitations of Reported Findings

Our original approach on the development of the Shop was to have it included in the main menu. From the main menu, the player would be able to access the Shop and purchase any of the items, given they have the correct number of coins. After some consideration, we realised that this approach would not work and we had to find a new way to implement it.

Another snag we hit was the ability to add new moves to the game and Shop. These new moves would function in the same way as the Flip move does, although they would have different effects. As mentioned earlier, the flip move was something that already existed in the code through FlipLevelTransformer and we added the ability for the player to use it in game. It would be possible to create more level transformers to be used in game, however due to how the game was coded, making these new transformers would take a significant amount of time and effort. Due to these issues, making new transformers may not be worth it.

Lessons Learned

Throughout our work in the enhancement proposal we came across various different ways that we could have implemented the Shop. Some of the approaches we considered were far too complicated, and some were lacking features we wanted. However, what we learned during our work was that there were many more approaches that we did not consider much, or even at all. There is very likely a way to implement our Shop, and future features, such that the number of affected subsystems is reduced.

Another lesson we learned through considering a wide variety of implementations is that NFRs do not always make the process of choosing an implementation easier. Some approaches may have advantages and disadvantages of similar magnitude as a completely separate approach, even if the two approaches do not share the NFRs for their pros and cons. For example, an approach may have more functionality, but would also require more work to implement while another approach might be easier to implement but might be unstable or of lower quality.

Another thing we have learned is that sometimes feasibility is the most important requirement. If we start to focus on an implementation that has a low feasibility, it could very easily end up as wasted time and effort. The feasibility of an approach is one of the first things that should be considered before putting too much time in it, and was in fact one of the major reasons why we did not initially try to follow the Persistent Shop approach.

References

[1] Tobias Markus.
"Obstacles".
Accessed December 1, 2017.
https://github.com/SuperTux/supertux/wiki/Obstacles.

[2] Wikipedia.
"Singleton Pattern Wikipedia Page."
Accessed November 27, 2017.
https://en.wikipedia.org/wiki/Singleton_pattern.

Dictionary

Player Status - An object managed by SuperTuxs Game Logic systems to store information about the current state of the player. It stores information such as which bonus the player currently has and how many coins they possess.

Shop - The system which we have proposed to implement.
The name of the object which manages the purchasing, retention, and selection of Shop Items.

Shop Item - An item made purchasable in the Shop. Could refer to a cosmetic, a sound, a move, or a bonus.
The name of the object which is stored in a collection by the Shop, used to manage information about which items the player has bought.

Shop Move - A special move made purchasable in the Shop.
The button used to trigger the Shop Move can be set in the appropriate input configuration menu. In game, if the Shop Move is used, it will perform some action such as inverting everything in the level or allowing Tux to jump a second time during a normal jump.

Shop Menu - The menu, which is available through the world map menu, where the player can purchase Shop Items from the Shop.
The object which is responsible for receiving requests from SuperTuxs menu system, making purchases and selections in the Shop, and applying changes to the Player Status object such as charging the player for purchases or giving Tux a bonus.