Project Phase 2

The firmware and app program description

Launch the application program when the cartridge is entered

Once the firmware is initialized, the display will indicate "OS started." After inserting the cartridge will launch our application and show a pink square to appear in the top left corner of the screen.

Read input from the controller

Any input from the controller will result in the movement of the displayed pink square by the "wadx" controls. Additionally, pressing the "cmd" button will alter the icon's shape and color.

A periodic timer

A countdown timer is integrated into the main function of the cartridge's code. The countdown timer is designed to check the status of the controller each time the countdown reaches zero.

Draw something in graphics mode

Within the cartridge, a pink square is rendered in graphics mode. This square can be moved using the W, A, D, and X keys from the controller.

Respond to commands and video interrupts

Command Button interrupt The command button functionality is configured to change the visual representation on the screen. Upon pressing the command button, the pink square transitions into a white circle.

Video Interrupt Video interrupts are managed within the firmware's interrupt handler. We have enabled the video interrupt handling and ensured that any pending video interrupts are cleared as they occur. After the firmware installation, it's possible to monitor and display the count of video interrupts that have been processed.

What we have changed to the API

Since the completion of project phase 1, our understanding of memory allocation and graphics of the RISC-V console has improved. For the implementation of **`allocateMemory`** and **`freeMemory`** APIs, we can leverage the **`malloc`** and **`free`** functions, provided we make necessary adjustments to the **`_sbrk`** function in our code.

Regarding the Graphics API provided by **`graphics.h`**, our improved understanding of the Video Controller Model has enabled us to refine it. We've upgraded the graphics API to allow switching between text mode and graphics mode. In text mode, programmers can first initialize a text buffer using **`createTextBuffer`** and subsequently obtain an index ID for this buffer. With this index ID, programmers can set the desired text display and its color using **`setTextBuffer`**. On the other hand, in graphics mode, a graphics buffer can be created using **`createGraphicBuffer`**, which returns an index ID for the buffer. Using this ID, programmers can define the dimensions and color for their graphics using **`setGraphicBuffer`**.

We have also made several enhancements to our API. Firstly, we've introduced a timer. Feedback from project phase 1 highlighted the need for a timer, especially for game developers who wanted to track and record time. Programmers can initiate the timer using **`startTimer`** and halt it with **`stopTimer`**. To reset the timer, **`resetTimer`** is available.

Our second improvement is the capability to save game progress. Feedback from project phase 1 suggested that enabling programmers to save and retrieve their game progress would be valuable. In response, we've added **`saveGameState`**, which allows game states to be saved. When using **`saveGameState`**, the game's state filename and the current game state (a structured format) must be provided. This structure offers flexibility, allowing programmers to define what aspects of the game are saved. For instance, within this structure, details like the player's name and experience points (XP) can be specified. To retrieve saved game states, **`loadGameState`** is available. Additionally, we have introduced **`listGameState`**, which returns a list of previously saved game states.

We are confident that these API improvements will empower programmers to create richer and more immersive games. In the following section, we detail the changes and enhancements to our API. A complete API list is provided at the end of this document.

Changed API

Name

- `createTextBuffer` – Allocates a buffer with specified text
- `createGraphicBuffer` – Allocates a buffer with specified dimensions
- `setTextBuffer` – Copies the contents of a text buffer to a specified position
- `setGraphicBuffer` – Add content to the graphic buffer
- `freeBuffer` – Deallocates a buffer
- `startTimer` – Start recording time
- `stopTimer` – Stop recording time and calculate the elapsed time
- `resetTimer` – Reset the timer to zero
- `saveGameState` – Save the current game state
- `loadGameState` – Load a saved game state from a file
- `listGameState` – List saved game states

Full API List

Name
`createThread` – Creates a new thread to run a specific function

Signature
```
#include "thread.h"
int createThread(void (*function)(void *), void *args);
```

Parameters
- `void (*function)(void *)` – A pointer to the function that the thread will execute
- `void *args` – A pointer to arguments that will be passed to the function

Return Value
On success, returns a unique thread ID. On failure, returns -1

Name
`terminateThread` – Terminates a thread specified by its thread ID

Signature
```
#include "thread.h"
int terminateThread(int threadID);
```

Parameters
- `int threadID` – The ID of the thread to be terminated

Return Value
On success, returns 0. On failure, returns -1

Name
`joinThread` – Waits for a specific thread to terminate

Signature
```
#include "thread.h"
int joinThread(int threadID);
```

Parameters

- `int threadID` – The ID of the thread to wait for

Return Value
On success, returns 0. On failure, returns -1

Name
`mutexLock` – Acquires a lock on a mutex to ensure synchronization

Signature
```
#include "thread.h"
int mutexLock(int mutexID);
```

Parameters
- `int mutexID` – The ID of the mutex to be locked

Return Value
On success, returns 0. On failure, returns -1

Name
`mutexUnlock` – Releases a lock on a mutex

Signature
```
#include "thread.h"
int mutexUnlock(int mutexID);
```

Parameters
- `int mutexID` – The ID of the mutex to be unlocked

Return Value
On success, returns 0. On failure, returns -1

Name
`getThreadStatus` – Queries the status of a thread

Signature
```
#include "thread.h"
```

```
int getThreadStatus(int threadID);
```

Parameters

- `int threadID` – The ID of the thread to query

Return Value

Returns the status of the thread (e.g., running, terminated). On failure, returns -1

Name

`triggerEvent` – Triggers event with a given event name

Signature

```
#include "event.h"
int triggerEvent(ControllerID controller, EventType event, void*
eventData);
```

Parameters

- `ControllerID controller` – The ID or reference to the specific controller for which the event is being triggered
- `EventType event` – The type of event that is being triggered
- `void* eventData` – A pointer to the event-specific data that will be passed to the registered callbacks

Return Value

Return 0 when success or -1 when occurs failure such as no callbacks are registered for the event

---

Name

`allocateMemory` – Allocates a block of memory of the specified size

Signature

```
#include "memoryManager.h"
void* allocateMemory(size_t size);
```

Parameters

- `size_t size` – The size of the memory block to be allocated

Return Value

On success, returns a pointer to the allocated memory block. Upon failure, returns NULL

Name

`freeMemory` – Deallocate a previous allocated block of memory

Signature

```
#include "memoryManager.h"
void freeMemory(void* ptr);
```

Parameters

- `void* ptr` – Pointer to the memory block that needs to be deallocated

Return Value

No return value

Name

`getWindowSize` – Get the width and height of the display window

Signature

```
#include "graphics.h"
struct windowSize {
   int width;
   int height;
};
windowSize getWindowSize(int windowId);
```

Parameters

- `int windowId` – The id of the input window

Return Value

- `windowSize` – The struct that contains the width and height of the window

---

Name

`createTextBuffer` – Allocates a buffer with specified text

Signature

```
#include "graphics.h"
```

```
int createTextBuffer(int length);
```

Parameters

- `int length` – The length of the text buffer

Return Value

Returns an ID for the Buffer representing the newly allocated buffer. Returns -1 on failure

---

Name

`createGraphicBuffer` – Allocates a buffer with specified dimensions

Signature

```
#include "graphics.h"
int createGraphicBuffer(int width, int height);
```

Parameters

- `int width, height` – Dimensions of the buffer

Return Value

Returns an ID for the Buffer representing the newly allocated buffer. Returns -1 on failure

---

Name

`setTextBuffer` – Copies the contents of a text buffer to a specified position

Signature

```
#include "graphics.h"
void setTextBuffer(int bufferId, char* input, int rgb);
```

Parameters

- `int bufferId` – The buffer to copy from
- `char* input` – The array that contains the characters for the content
- `int rgb` – The color of the text, integer starts from 1 to 256

Return Value

No return value

Name
`setGraphicBuffer` – Add content to the graphic buffer

Signature
```
#include "graphics.h"
void setGraphicBuffer(int bufferId, int* input, int rgb);
```

Parameters
- `int bufferId` – The buffer to copy from
- `int* input` – The array that contains the positions of the pixels
- `int rgb` – The color of the pixels, integer starts from 1 to 256

Return Value
No return value

Name
`freeBuffer` – Deallocates a buffer

Signature
```
#include "graphics.h"
void freeBuffer(int bufferId);
```

Parameters
- `int bufferId` – The buffer to deallocate

Return Value
No return value

Name
`startTimer` – Start recording time

Signature
```
#include "timeRecorder.h"
void startTimer();
```

Parameters

Return Value
No return value

Name
`stopTimer` – Stop recording time and calculate the elapsed time

Signature
```
#include "timeRecorder.h"
double stopTimer();
```

Parameters

Return Value
Returns the elapsed time in seconds as a double value

Name
`resetTimer` – Reset the timer to zero

Signature
```
#include "timeRecorder.h"
void resetTimer();
```

Parameters

Return Value

Name
`saveGameState` – Save the current game state

Signature
```
#include "gameState.h"
void saveGameState(const char* filenmae, GameState* gameState);
```

Parameters
- `const char* filename` – The name of the file where the game state will be saved
- `GameState* gameState` – A pointer to the game state structure to be saved

Return Value
Returns 0 when successful and a non-zero value on failure, such as file I/O errors

---

Name
`loadGameState` – Load a saved game state from a file

Signature
```
#include "gameState.h"
void loadGameState(const char* filename, GameState* gameState);
```

Parameters
- `const char* filename` – The name of the file from which the game state will be loaded
- `GameState* gameState` – A pointer to the game state structure where the loaded state will be stored

Return Value
Returns 0 when successful and a non-zero value on failure, such as file I/O errors

---

Name
`listGameState` – List saved game states

Signature
```
#include "gameState.h"
char** listGameState();
```

Parameters

Return Value
Returns a pointer for to the array that contains the list of game states