ReadMe User

This contains all the functions within my game.

class Application2D : public aie::Application {

public:

//Constructor

Application2D();

// Destructor

virtual ~Application2D();

//Startup

virtual bool startup();

//Shutdown

virtual void shutdown();

// Update via deltatime

virtual void update(float deltaTime);

//draws the objects.

virtual void draw();

GSM \*getGSM() { return gsm; }

aie::Renderer2D\* m\_2dRenderer;

protected:

GSM \*gsm;

};

// This is based off of example code given to us by our teacher, Jeff Cotter.

// I am using this becuase it provides a more graceful way to allow throws and asserts to be done.

#define eTHROW( errormessage ) do { \

std::ostringstream oss; \

oss << errormessage; \

/\*Remove file path, so we end up with just the file name.\*/ \

const char \*name = strrchr( \_\_FILE\_\_, '\\' ); \

if (name) \

{ \

name++; \

} \

else if (name == 0) \

{ \

/\*We may be on an OS that uses the alternate slash for paths.\*/ \

name = strrchr(\_\_FILE\_\_, '/'); \

if (name) name++; \

} \

if (name == 0) name = \_\_FILE\_\_; \

std::string eMessage = "\"" + oss.str() + "\" File: " + name + ". Line: " + std::to\_string(\_\_LINE\_\_) + "."; \

throw std::runtime\_error(eMessage); } while(0)

#define eASSERT( condition ) do { if( !(condition) ) eTHROW( "Assertion Failure" ); } while(0)

/\*

\* class BinaryTree

\*

\* This is my BinaryTree class and is a customer container that will be used for m\_registeredstates as the states that are stored in my game.

\* We are using this since we doing care about the order of our registered states. we just want to check to see if we have a state of a certain name.

\*

\* @author - Zackary Direen, Academy of Interactive Entertainment, 2017

\*/

template <class T, class U>

class BinaryTree

{

/\*

\* struct TreeNode

\*

\* This is a TreeNode. This is what each item in the binaryTree is based off of.

\*

\* @author - Zackary Direen, Academy of Interactive Entertainment, 2017

\*/

struct TreeNode

{

T key;

U value;

TreeNode\* left;

TreeNode\* right;

TreeNode(T newKey, U newValue)

{

key = newKey;

value = newValue;

left = NULL;

right = NULL;

}

};

private:

TreeNode\* root;

/\* void Insert(T newKey, U newValue, TreeNode\* &theRoot)

\* This function allows for the insertion of a new treeNode.

\*

\*

\* @param1 newKey - the key that we want our value to have. this is for accessing it.

\*

\* @param2 newValue - The value stored within the location for the key.

\*

\* @param3 &theRoot - This is the root treenode of this tree.

\*

\* @returns void.

\*/

void Insert(T newKey, U newValue, TreeNode\* &theRoot)

{

if (theRoot == NULL)

{

theRoot = new TreeNode(newKey, newValue);

return;

}

if (newKey < theRoot->key)

{

Insert(newKey, newValue, theRoot->left);

}

else if(newKey > theRoot->key)

{

Insert(newKey, newValue, theRoot->right);

}

else

{

theRoot->value = newValue;

}

}

/\* void RemoveTree(TreeNode\* theRoot)

\* This function will delete a tree from theRoot downwards.

\*

\*

\*

\* @param1 &theRoot - This is the root treenode of this tree.

\*

\* @returns void.

\*/

void RemoveTree(TreeNode\* theRoot)

{

if (theRoot != NULL)

{

RemoveTree(theRoot->left);

RemoveTree(theRoot->right);

delete theRoot;

}

}

/\* U &Search(TreeNode \*theRoot, const T &key)

\* This function will search though the tree from the root and return the value within the key if found.

\*

\*

\*

\* @param1 \*theRoot - This is the root treenode of this tree.

\*

\* @param2 const T &key - This is the key of a treenode.

\*

\* @returns U - a value of a treeNode.

\*/

U &Search(TreeNode \*theRoot, const T &key)

{

if (!theRoot)

{

eTHROW("Tried to access a list that doesnt exist.");

}

if (theRoot->key == key)

{

return theRoot->value;

}

else if (key < theRoot->key)

{

return Search(theRoot->left, key);

}

else if (key > theRoot->key)

{

return Search(theRoot->right, key);

}

else

{

eTHROW("Tried to access a value in the list that is not avalible.");

}

}

public:

/\*

\* BinaryTree()

\*

\* This is the BinaryTree constructor.

\*

\*/

BinaryTree()

{

root = NULL;

}

/\* void AddItem(T newKey, U newValue)

\* This function will call the insert function with the parameters passed.

\*

\*

\*

\* @param1 newKey - This is the key of a treenode.

\*

\* @param2 U - a value of a treeNode.

\*

\* @returns void.

\*/

void AddItem(T newKey, U newValue)

{

Insert(newKey, newValue, root);

}

/\* void U &operator [] (T newKey)

\* This function will call the search function with the parameters passed.

\*

\*

\*

\* @param1 newKey - This is the key of a treenode.

\*

\*

\* @returns U - a value of a treeNode.

\*/

U &operator [] (T newKey)

{

return Search(root, newKey);

}

/\* void clear()

\* This function will call the RemoveTree functiopn from the root onwards, deleting the whole tree, then sets the root to nullptr.

\*

\* @returns void.

\*/

void clear()

{

RemoveTree(root);

root = nullptr;

}

};

// Forward declare to let us use the type in this namespace

class Application2D;

class IState;

/\*

\* class GSM

\*

\* this is the game state manager. this is responible for handling all the states within the game and registering, pushing and popping states into the registered states (which is a binaryTree)

\* and the active states, which is the Doubly linked list.

\*

\* @author - Zackary Direen, Academy of Interactive Entertainment, 2017

\*/

class GSM {

public:

/\*

\* GSM();

\*

\* This is the GSM constructor.

\*

\*/

GSM();

/\*

\* ~GSM();

\*

\* This is the GSM destructor.

\*

\*/

~GSM();

/\* void updateStates(float deltaTime);

\* This function is responsible for updating the states based on which states are in use and which are just registered.

\*

\* parameter 1 deltaTime - Delta Time.

\* @returns void

\*/

void updateStates(float deltaTime);

/\* void renderStates();

\* This function is responsible for rendering the states based on which states are in use and which are just registered.

\*

\* parameter 1 deltaTime - Delta Time.

\* @returns void

\*/

void renderStates();

/\* void registerState(int ID, IState\* state);

\* This function is responsible for registering a state into the registered states with an ID and the state itself.

\*

\* parameter1 ID - the ID of the state to be registered.

\*

\* parameter2 state - The state we are trying to register.

\*

\* @returns void

\*/

void registerState(int ID, IState\* state);

/\* void pushState(int ID);

\* This function is responsible for pushing a state into the active states with an ID.

\*

\* parameter1 ID - the ID of the state to be pushed.

\*

\*

\* @returns void

\*/

void pushState(int ID);

/\* void popState();

\* This function is responsible for popping a state from the active states to the registered states.

\*

\* parameter1 ID - the ID of the state to be registered.

\*

\*

\* @returns void

\*/

void popState();

/\* IState\* getTopState();

\* This function is responsible for getting the first state in the linked list of active states and for the purposes of running that state.

\*

\*

\*

\* @returns IState\* - the top state in the list to be run.

\*/

IState\* getTopState();

private:

// This is using a Binary tree for the registered states.

BinaryTree<int, IState\*> m\_registeredStates;

// And uses a linked list for the active states.

List<IState\*> m\_activeStates;

// This is an enum class that will have the 3 commands we want.

enum class ECommand { REGISTER, PUSH, POP };

struct ICommand

{

ECommand c\_command;

int c\_id;

IState\* c\_state;

};

List<ICommand> m\_commands;

// this command will happen at the beginning of the update to be able to swap states when needed.

/\* void processCommands();

\* This function is responsible for processing the pushing, popping and registering of each state.

\*

\*

\*

\* @returns void

\*/

void processCommands();

/\* void proceesRegisterState(int ID, IState\* state);

\* This function is responsible for calling the registerState function with the parameters provided.

\*

\*

\* parameter1 ID - the ID of the state to be registered.

\*

\* parameter2 state - The state we are trying to register.

\*

\* @returns void

\*/

void proceesRegisterState(int ID, IState\* state);

/\* void processPushState(int ID);

\* This function is responsible for calling the pushState function with the parameters provided.

\*

\*

\* parameter1 ID - the ID of the state to be registered.

\*

\*

\* @returns void

\*/

void processPushState(int ID);

/\* void processPopState();

\* This function is responsible for calling the popState function.

\*

\*

\*

\*

\* @returns void

\*/

void processPopState();

};

template<typename T>

class List;

template<typename T>

class ListIterator;

/\*

\*class ListNode

\* This is a ListNode, it is a object within my linked list.

\* @author - Zackary Direen, Academy of Interactive Entertainment, 2017

\*

\*/

template<typename T>

class ListNode

{

private:

// Only a list of the same type will be able use these ListNodes.

friend List<T>;

friend ListIterator<T>;

ListNode<T>\* pNext;

ListNode<T>\* pPrev;

/\* ListNode<T>()

\*

\* This is a default constuctor for ListNode.

\*

\*/

ListNode<T>() { pNext = nullptr; pPrev = nullptr; }

// Value can now be of whatever type

T value;

};

/\*

\*class ListIterator

\* This is a ListIterator, it this is a part of my linked list that allows me to iterate though it.

\* @author - Zackary Direen, Academy of Interactive Entertainment, 2017

\*

\*/

template<typename T>

class ListIterator

{

public:

/\* ListIterator()

\*

\* This is a default constuctor for ListIterator

\*

\*/

ListIterator() { nodePtr = 0; };

/\* ListIterator(ListNode<T> \*ptr)

\*

\* This is a constructor that is given a node pointer.

\*

\*/

ListIterator(ListNode<T> \*ptr) { nodePtr = ptr; };

/\* ListIterator(const ListIterator<T> &iter)

\*

\* This is a constructor that is a previous list iterator and sets the nodePtr to the current one.

\*

\*/

ListIterator(const ListIterator<T> &iter) { nodePtr = iter.nodePtr; };

//ListIterator &operator = (const ListIterator<T> &iter) { nodePtr = iter.nodePtr; return \*this; };

// converting an iterator to Node\*

operator ListNode<T>\*() const { return nodePtr; }

/\* bool operator !=(const ListIterator& x)

\*

\* This is the function responsible for testing two iterators to see if they are not equal.

\*

\*

\* @returns bool - True or false.

\*/

bool operator !=(const ListIterator& x) const { return nodePtr != x.nodePtr; }

/\* bool operator ==(const ListIterator& x)

\*

\* This is the function responsible for testing two iterators to see if they are equal.

\*

\*

\* @returns bool - True or false.

\*/

bool operator ==(const ListIterator& x) const { return nodePtr == x.nodePtr; }

/\* ListIterator& operator++()

\*

\* This is the function responsible for returning the next index of the ListIterator.

\*

\*

\* @returns this - a ListIterator.

\*/

ListIterator& operator++() { nodePtr = nodePtr->pNext; return \*this; }

// decrement operator

/\* ListIterator& operator++()

\*

\* This is the function responsible for returning the previous index of the ListIterator.

\*

\*

\* @returns this - a ListIterator.

\*/

ListIterator& operator--() { nodePtr = nodePtr->pPrev; return \*this; }

// This function allows the Iternator to iterate to a certain position.

// It will return the nodePtr that is at that position.

/\* ListIterator& operator+=(int position)

\*

\* This is the function responsible for incrementing the ListIternator until the position given as a parameter.

\*

\*

\* @parameter 1 position - a number given for the ListIterator to iternate to.

\*

\* @returns this - a ListIterator.

\*/

ListIterator& operator+=(int position) {

for (int i = 0; i < position; i++)

{

nodePtr = nodePtr->pNext;

return \*this;

}

return \*this;

}

/\* ListIterator& operator++(int)

\*

\* This is the function responsible for incrementing the ListIternator of itself.

\*

\*

\* @parameter 1 int

\*

\* @returns this - a ListIterator.

\*/

ListIterator& operator++(int) { nodePtr = nodePtr->pNext; return \*this; }

/\* T& operator\*()

\*

\* This is the function responsible for returning the value associated with iterator.

\*

\*

\*

\* @returns T - a value.

\*/

T& operator\*() const { return nodePtr->value; }

ListNode<T>\* nodePtr;

/\* ~ListIterator()

\*

\* This is a default destructor for ListIterator

\*

\*/

~ListIterator() {};

private:

};

/\*

\*class List

\* This is a List, it this is my doubly linked listed in its entirety, comprised of ListNodes and the iterator.

\* @author - Zackary Direen, Academy of Interactive Entertainment, 2017

\*

\*/

template<typename T>

class List

{

public:

int listLength;

typedef ListIterator<T> interator;

/\* List()

\*

\* This is a default constructor for List

\*

\*/

List()

{

tail = head = nullptr;

listLength = 0;

}

/\* ~List()

\*

\* This is a default destructor for ListIterator

\*

\*/

~List()

{

}

// This will create a new list node and put a value in it, then it will put it at the front of the list.

// It takes a value and returns void.

/\* void pushFront(T value)

\*

\* This is the function responsible for creating a new list node and put a value in it, then it will put it at the front of the list.

\*

\*

\* @parameter 1 value - something that wants to be sorted in the list.

\*

\*

\* @returns void

\*/

void pushFront(T value)

{

ListNode<T> \*ptr = new ListNode<T>;

ptr->value = value;

if (this->listLength == 0)

{

head = ptr;

tail = ptr;

}

else

{

ptr->pNext = head;

ptr->pPrev = nullptr;

head->pPrev = ptr;

head = ptr;

}

listLength++;

}

/\* void pushBack(T value)

\*

\* This is the function responsible for creating a new list node and put a value in it, then it will put it at the back of the list.

\*

\*

\* @parameter 1 value - something that wants to be sorted in the list.

\*

\*

\* @returns void

\*/

void pushBack(T value)

{

ListNode<T> \*ptr = new ListNode<T>();

ptr->value = value;

if (this->listLength == 0)

{

head = ptr;

tail = ptr;

}

else

{

ptr->pNext = nullptr;

ptr->pPrev = tail;

tail->pNext = ptr;

tail = ptr;

}

listLength++;

}

/\* void popFront()

\*

\* This is the function responsible for popping a list node at the front of the list.

\*

\*

\* @returns void

\*/

void popFront()

{

ListNode<T> \*ptr = head->pNext;

delete head;

ptr->pPrev = nullptr;

head = ptr;

listLength--;

}

/\* void popBack()

\*

\* This is the function responsible for popping a list node at the back of the list.

\*

\*

\* @returns void

\*/

void popBack()

{

ListNode<T> \*ptr = tail->pPrev;

delete tail;

if (ptr)

{

ptr->pNext = nullptr;

tail = ptr;

}

else

{

head = tail = nullptr;

}

listLength--;

}

// this function should insert a value before the node parameter.

/\* void insertBefore(int position, T insertedValue)

\*

\* This is the function responsible for inserting a value before the node parameter. and if the list's size is 0, it will pushFront and if its

\* at the end of the list, it will pushBack.

\*

\* @parameter 1 posiiton - a position in the linked list.

\*

\* @parameter 2 insertedValue - a value that wants to be inserted.

\*

\* @returns void

\*/

void insertBefore(int position, T insertedValue)

{

if (position == 0)

{

pushFront(insertedValue);

}

else if (position < 0 || position > listLength)

{

throw("Error: tried to enter a value outside of the list.");

}

else

{

List<T>::interator holder = begin();

holder += (position);

ListNode<T> \*ptr = new ListNode<T>;

ptr->pNext = (holder.nodePtr)->pNext;

ptr->pPrev = (holder.nodePtr);

(holder.nodePtr)->pNext->pPrev = ptr;

(holder.nodePtr)->pNext = ptr;

ptr->value = insertedValue;

head = ptr;

listLength++;

}

}

// this function should insert a value after the node parameter. it takes the position of the list and the value it wants inserted as parameters and returns void.

/\* void insertBefore(int position, T insertedValue)

\*

\* This is the function responsible for inserting a value after the node parameter. and if the list's size is 0, it will pushFront and if its

\* at the end of the list, it will pushBack.

\*

\* @parameter 1 posiiton - a position in the linked list.

\*

\* @parameter 2 insertedValue - a value that wants to be inserted.

\*

\* @returns void

\*/

void insertAfter(int position, T insertedValue)

{

if (position == 0)

{

pushFront(insertedValue);

}

else if (position == listLength)

{

pushBack(insertedValue);

}

else if (position < 0 || position > listLength)

{

throw("Error: tried to enter a value outside of the list.");

}

else

{

List<T>::interator holder = begin();

holder += (position);

ListNode<T> \*ptr = new ListNode<T>;

ptr->pNext = (holder.nodePtr);

ptr->pPrev = (holder.nodePtr)->pPrev;

(holder.nodePtr)->pPrev->pNext = ptr;

//(holder.nodePtr)->pNext = ptr;

ptr->value = insertedValue;

tail = ptr;

listLength++;

}

}

/\* void erase(int position)

\*

\* This will delete a listNode at a certain position and decrease the list. and if its at the front of the list it will popFront

\* and if its at the end it will popBack.

\*

\* @parameter 1 posiiton - a position in the linked list.

\*

\*

\* @returns void

\*/

void erase(int position)

{

if (position == 0)

{

popFront();

}

else if (position == listLength)

{

popBack();

}

else if (position < 0 || position > listLength)

{

throw("Error: tried to delete a value outside of the list.");

}

else

{

List<T>::interator holder = begin();

holder += (position);

(holder.nodePtr)->pNext->pPrev = (holder.nodePtr)->pPrev;

(holder.nodePtr)->pPrev->pNext = (holder.nodePtr)->pNext;

delete (holder.nodePtr);

listLength--;

}

}

/\* void erase(int position)

\*

\* This will delete a value within the list dependant on what the value is if it is found.

\*

\* @parameter 1 value - the value of a object in the linked list.

\*

\*

\* @returns void

\*/

void remove(T value)

{

ListNode<T> \*ptr = head->next;

while (ptr != head) {

Node \*next = ptr->next;

if (ptr->value == value) {

ptr->prev->next = p->next;

ptr->next->prev = p->prev;

delete p;

}

p = next;

}

}

/\* void deleteList()

\*

\* This is responsible for deleting the entire list.

\*

\*

\* @returns void

\*/

void deleteList()

{

ListNode<T> \*pDel = head;

while (pDel != NULL)

{

head = head->pNext;

delete pDel;

pDel = head;

}

tail = head = NULL;

listLength = 0;

}

ListIterator<T> List<T>::begin()

{

return ListIterator<T>(head);

}

ListIterator<T> List<T>::end()

{

return ListIterator<T>();

}

// This returns the value of the first value in the first item of the list.

T& first()

{

return head->value;

}

// This returns the value of the last value in the last item of the list.

T& last()

{

return tail->value;

}

protected:

ListNode<T> \*head;

ListNode<T> \*tail;

};

template <class T>

class Stack {

public:

T object; // this is to store values.

T tempTop; // this is to store the value of the top of the stack when it is deleted.

int top; // this is a index of the amount in the stack.

T\* myArray;

int size;

/\* Stack()

\*

\* This is a default constructor for Stack

\*

\*/

Stack()

{

}

/// This is the constructor. we do not use the default constructor, as we always want this to have a size.

/\* Stack(int n)

\*

\* This is a constructor for List that will take a size parameter.

\*

\*

\* parameter 1 n - the amount of items you want in the stack.

\*

\*/

Stack(int n)

{

size = n;

eASSERT(n > 0);

// this creates an empty stack with the amount of spaces previously defined by 'default\_value'

myArray = new T[n];

// this defaults the index to be -1 to signifiy the stack is empty of values.

top = -1;

}

/\* ~Stack()

\*

\* This is a default destructor for Stack

\*

\*/

~Stack()

{

}

/\* void push(T object)

\*

\*This pushes a value to the top of the stack and will increment the index.

\*

\*

\* @parameter 1 object - a value you want to push into the stack.

\*

\*

\* @returns void

\*/

void push(T object)

{

top++;

if (top < size)

{

myArray[top] = object;

}

// if the stack is full, dont allow the user to add to the stack

else

{

eTHROW("You tried to push the stack above its set size.");

}

}

/\* T pop()

\*

\* This pops a value from the top of the stack and decrement the stack.

\*

\* @returns T - the object that is popped.

\*/

T pop()

{

if (top == -1)

{

eTHROW("You tried to pop from the a list that is empty.");

}

else

{

T data = myArray[top];

myArray[top] = NULL;

top--;

return data;

}

}

/\* T pop()

\*

\* This function is responsible for checking to see if a stack is empty or not.

\* @returns bool - true or false.

\*/

bool empty()

{

if (top == -1)

{

return true;

}

else

{

return false;

}

}

protected:

};

/\*\* This state is an abstract class interface that describes how

other class objects will interact with it - Cannot be instantiated, must

have a child class inherit from it to use\*/

class Application2D;

class GSM;

namespace aie {

class Renderer2D;

class Font;

}

/\*

\* class IState

\*

\* this is the blueprint of each of the states within the game and what all states will have in my game.

\*

\* @author - Zackary Direen, Academy of Interactive Entertainment, 2017

\*/

class IState {

public:

IState(Application2D \*\_app, GSM \*\_gsm) : app(\_app), gsm(\_gsm) {};

/\* virtual ~IState() = default;

\* This is the virtual destructor for every state.

\* This is called when the main program has reached the end of its scope and this will be called on every state.

\*/

virtual ~IState() = default;

// These are pure virtual functions, all the other states have their own update and render functions that will take over these functions.

/\* virtual void update(float dt) = 0;

\* This function is a pure virtual function that every state should have. This is the update function, which will update every frame.

\*

\* parameter 1 dt - Delta Time.

\* @returns void.

\*/

virtual void update(float dt) = 0;

/\* virtual void render() = 0;

\* This function is a pure virtual function that every state should have. This is the render function, which will draw to the screen based on when update happens.

\*

\*

\* @returns void.

\*/

virtual void render() = 0;

protected:

aie::Font \*m\_font;

Application2D \*app;

GSM \*gsm;

};

namespace aie {

class Font;

class Renderer2D;

}

class PauseState : public IState

{

public:

PauseState(Application2D \*\_app, GSM \*\_gsm);

~PauseState();

/\* virtual void update(float dt);

\* This function is a virtual function that is responsible for updating this state every frame.

\*

\* parameter 1 dt - Delta Time.

\* @returns void

\*/

virtual void update(float dt);

/\* virtual void render();

\* This function is a virtual function that is responsible for drawing the items in this state every frame based on the update function.

\*

\* @returns void

\*/

virtual void render();

private:

std::unique\_ptr<aie::Font> m\_font;

Object\* paddleLeft;

Object\* paddleRight;

Object\* Ball;

aie::Input \*input;

};

/\*

\* class SplashState

\*

\* this is SplashState and is the first state of my game. this is based on the IState.

\*

\* @author - Zackary Direen, Academy of Interactive Entertainment, 2017

\*/

class SplashState :

public IState {

public:

SplashState(Application2D \*\_app, GSM \*\_gsm);

/\* virtual ~SplashState();

\* This function is a virtual destructor that should exist so that the state has a destructor.

\*

\*

\*/

virtual ~SplashState();

/\* virtual void update(float dt);

\* This function is a virtual function that is responsible for updating this state every frame.

\*

\* parameter 1 dt - Delta Time.

\* @returns void

\*/

virtual void update(float dt);

/\* virtual void render();

\* This function is a virtual function that is responsible for drawing the items in this state every frame based on the update function.

\*

\* @returns void

\*/

virtual void render();

private:

char \*SplashText;

const float delayTime = 0.3f;

float switchStateTimer;

void updateStateTimer(float dt);

aie::Input \*input;

};

namespace aie {

class Font;

class Renderer2D;

}

/\*

\* class MenuState

\*

\* this is MenuState and is the second state of my game. this is based on the IState.

\*

\* @author - Zackary Direen, Academy of Interactive Entertainment, 2017

\*/

class MenuState : public IState

{

public:

MenuState::MenuState(Application2D \*\_app, GSM \*\_gsm);

~MenuState();

/\* virtual void update(float dt);

\* This function is a virtual function that is responsible for updating this state every frame.

\*

\* parameter 1 dt - Delta Time.

\* @returns void

\*/

virtual void update(float dt);

/\* virtual void render();

\* This function is a virtual function that is responsible for drawing the items in this state every frame based on the update function.

\*

\* @returns void

\*/

virtual void render();

private:

std::unique\_ptr<aie::Font> m\_font;

float switchStateTimer;

void updateStateTimer(float dt);

int WhatsSelected, maxMenuOption;

aie::Input \*input;

};

/\*

\* class LoadState

\*

\* this is LoadState and is the third state of my game. this is based on the IState.

\*

\* @author - Zackary Direen, Academy of Interactive Entertainment, 2017

\*/

class LoadState :

public IState {

public:

LoadState(Application2D \*\_app, GSM \*\_gsm);

virtual ~LoadState();

/\* virtual void update(float dt);

\* This function is a virtual function that is responsible for updating this state every frame.

\*

\* parameter 1 dt - Delta Time.

\* @returns void

\*/

virtual void update(float dt);

/\* virtual void render();

\* This function is a virtual function that is responsible for drawing the items in this state every frame based on the update function.

\*

\* @returns void

\*/

virtual void render();

private:

char \*loadText;

char \*playerControls;

const float delayTime = 0.3f;

float switchStateTimer;

void updateLoadText(float dt);

void updateStateTimer(float dt);

aie::Font \*m\_instructFont;

};