

Wiesn-Run

Erzeugt von Doxygen 1.8.6

Son Jul 12 2015 21:30:48

Inhaltsverzeichnis

1	Ausstehende Aufgaben	1
2	Hierarchie-Verzeichnis	3
2.1	Klassenhierarchie	3
3	Klassen-Verzeichnis	5
3.1	Auflistung der Klassen	5
4	Datei-Verzeichnis	7
4.1	Auflistung der Dateien	7
5	Klassen-Dokumentation	9
5.1	Audio Klassenreferenz	9
5.1.1	Ausführliche Beschreibung	10
5.1.2	Beschreibung der Konstruktoren und Destruktoren	10
5.1.2.1	Audio	10
5.1.2.2	~Audio	10
5.1.3	Dokumentation der Elementfunktionen	10
5.1.3.1	getSample	10
5.1.3.2	getSamplenumber	11
5.1.3.3	getSource	11
5.1.3.4	normalize	11
5.1.3.5	readSamples	11
5.1.3.6	to16bitSample	11
5.2	AudioControl Klassenreferenz	12
5.2.1	Ausführliche Beschreibung	14
5.2.2	Beschreibung der Konstruktoren und Destruktoren	14
5.2.2.1	AudioControl	14
5.2.2.2	~AudioControl	14
5.2.3	Dokumentation der Elementfunktionen	14
5.2.3.1	instancepaCallback	14
5.2.3.2	playInitialize	15
5.2.3.3	playTerminate	15

5.2.3.4	staticpaCallback	15
5.2.3.5	updatePlayevents	15
5.3	audioCooldownstruct Strukturreferenz	16
5.4	audioCooldownStruct Strukturreferenz	16
5.5	audioDistanceStruct Strukturreferenz	17
5.6	audioStruct Strukturreferenz	18
5.7	collisionStruct Strukturreferenz	18
5.7.1	Ausführliche Beschreibung	18
5.8	compareGameObjects Strukturreferenz	18
5.8.1	Ausführliche Beschreibung	18
5.9	compareScores Strukturreferenz	19
5.9.1	Ausführliche Beschreibung	19
5.10	Enemy Klassenreferenz	19
5.10.1	Beschreibung der Konstruktoren und Destruktoren	20
5.10.1.1	Enemy	20
5.10.2	Dokumentation der Elementfunktionen	21
5.10.2.1	getDeath	21
5.10.2.2	getDeathCooldown	21
5.10.2.3	getFireCooldown	21
5.10.2.4	getHealth	21
5.10.2.5	getInflictedDamage	21
5.10.2.6	setDeath	21
5.10.2.7	setHealth	22
5.11	Game Klassenreferenz	22
5.11.1	Ausführliche Beschreibung	25
5.11.2	Beschreibung der Konstruktoren und Destruktoren	25
5.11.2.1	Game	25
5.11.3	Dokumentation der Elementfunktionen	26
5.11.3.1	appendWorldObjects	26
5.11.3.2	calculateMovement	26
5.11.3.3	detectCollision	26
5.11.3.4	displayStatistics	26
5.11.3.5	endGame	27
5.11.3.6	evaluateInput	27
5.11.3.7	eventFilter	27
5.11.3.8	exitGame	27
5.11.3.9	getStepIntervall	28
5.11.3.10	handleCollisions	28
5.11.3.11	loadLevelFile	28
5.11.3.12	menuInit	28

5.11.3.13	reduceWorldObjects	28
5.11.3.14	renderGraphics	29
5.11.3.15	setState	29
5.11.3.16	start	29
5.11.3.17	startNewGame	30
5.11.3.18	step	30
5.11.3.19	timerEvent	30
5.11.3.20	updateAudioevents	31
5.11.3.21	updateHighScore	31
5.11.3.22	updateScore	31
5.11.4	Dokumentation der Datenelemente	31
5.11.4.1	levelSpawn	31
5.11.4.2	stepCount	31
5.12	GameObject Klassenreferenz	31
5.12.1	Beschreibung der Konstruktoren und Destruktoren	32
5.12.1.1	GameObject	32
5.13	Input Klassenreferenz	33
5.13.1	Ausführliche Beschreibung	34
5.13.2	Beschreibung der Konstruktoren und Destruktoren	35
5.13.2.1	Input	35
5.13.2.2	~Input	35
5.13.3	Dokumentation der Elementfunktionen	35
5.13.3.1	evaluatekeyEvent	35
5.13.3.2	getKeyactions	35
5.13.3.3	getKeyletters	36
5.13.3.4	getLastKeyaction	36
5.13.3.5	getLastKeyletter	36
5.13.3.6	updateKeys	36
5.14	Menu Klassenreferenz	37
5.14.1	Ausführliche Beschreibung	38
5.14.2	Beschreibung der Konstruktoren und Destruktoren	38
5.14.2.1	Menu	38
5.14.3	Dokumentation der Elementfunktionen	38
5.14.3.1	addEntry	38
5.14.3.2	changeSelection	39
5.14.3.3	clear	39
5.14.3.4	displayInit	39
5.14.3.5	displayUpdate	39
5.14.3.6	getEntry	40
5.14.3.7	getSelection	40

5.14.3.8	getTitle	40
5.14.3.9	getType	41
5.14.3.10	selectFirstEntry	41
5.15	Menu::menuEntry Strukturreferenz	41
5.15.1	Ausführliche Beschreibung	41
5.16	MovingObject Klassenreferenz	42
5.16.1	Dokumentation der Elementfunktionen	42
5.16.1.1	flipHorizontal	42
5.16.1.2	updatePosition	43
5.17	PaDeviceInfo Strukturreferenz	43
5.17.1	Ausführliche Beschreibung	43
5.17.2	Dokumentation der Datenelemente	43
5.17.2.1	defaultHighInputLatency	43
5.17.2.2	defaultLowInputLatency	43
5.17.2.3	hostApi	43
5.18	PaHostApiInfo Strukturreferenz	44
5.18.1	Ausführliche Beschreibung	44
5.18.2	Dokumentation der Datenelemente	44
5.18.2.1	defaultInputDevice	44
5.18.2.2	defaultOutputDevice	44
5.18.2.3	deviceCount	44
5.18.2.4	name	44
5.18.2.5	structVersion	44
5.18.2.6	type	44
5.19	PaHostErrorInfo Strukturreferenz	45
5.19.1	Ausführliche Beschreibung	45
5.19.2	Dokumentation der Datenelemente	45
5.19.2.1	errorCode	45
5.19.2.2	errorText	45
5.19.2.3	hostApiType	45
5.20	PaStreamCallbackTimeInfo Strukturreferenz	45
5.20.1	Ausführliche Beschreibung	46
5.20.2	Dokumentation der Datenelemente	46
5.20.2.1	currentTime	46
5.20.2.2	inputBufferAdcTime	46
5.20.2.3	outputBufferDacTime	46
5.21	PaStreamInfo Strukturreferenz	46
5.21.1	Ausführliche Beschreibung	46
5.21.2	Dokumentation der Datenelemente	46
5.21.2.1	inputLatency	47

5.21.2.2	outputLatency	47
5.21.2.3	sampleRate	47
5.21.2.4	structVersion	47
5.22	PaStreamParameters Strukturreferenz	47
5.22.1	Ausführliche Beschreibung	47
5.22.2	Dokumentation der Datenelemente	48
5.22.2.1	channelCount	48
5.22.2.2	device	48
5.22.2.3	hostApiSpecificStreamInfo	48
5.22.2.4	sampleFormat	48
5.22.2.5	suggestedLatency	48
5.23	Player Klassenreferenz	48
5.23.1	Beschreibung der Konstruktoren und Destrukturen	50
5.23.1.1	Player	50
5.23.2	Dokumentation der Elementfunktionen	50
5.23.2.1	decreaseAlcoholLevel	50
5.23.2.2	getAlcoholLevel	50
5.23.2.3	getAmmunatiuon	51
5.23.2.4	getFireCooldown	51
5.23.2.5	getHealth	51
5.23.2.6	getImmunityCooldown	51
5.23.2.7	getInflictedDamage	51
5.23.2.8	increaseAlcoholLevel	51
5.23.2.9	inJump	52
5.23.2.10	receiveDamage	52
5.23.2.11	setHealth	52
5.23.2.12	setImmunityCooldown	52
5.23.2.13	update	52
5.24	AudioControl::playStruct Strukturreferenz	52
5.25	PowerUp Klassenreferenz	53
5.25.1	Ausführliche Beschreibung	54
5.25.2	Beschreibung der Konstruktoren und Destrukturen	54
5.25.2.1	PowerUp	54
5.25.2.2	~PowerUp	54
5.25.3	Dokumentation der Elementfunktionen	54
5.25.3.1	getAlcoholLevelBonus	54
5.25.3.2	getAmmunationBonus	55
5.25.3.3	getHealthBonus	55
5.25.3.4	getImmunityCooldownBonus	55
5.25.3.5	getPowerUPType	55

5.26	RenderBackground Klassenreferenz	55
5.26.1	Ausführliche Beschreibung	56
5.26.2	Beschreibung der Konstruktoren und Destruktoren	56
5.26.2.1	RenderBackground	56
5.26.3	Dokumentation der Elementfunktionen	56
5.26.3.1	setPos	56
5.26.3.2	updateBackgroundPos	57
5.26.3.3	updateParallaxe	57
5.27	RenderGUI Klassenreferenz	57
5.27.1	Ausführliche Beschreibung	58
5.27.2	Beschreibung der Konstruktoren und Destruktoren	58
5.27.2.1	RenderGUI	58
5.27.3	Dokumentation der Elementfunktionen	58
5.27.3.1	setPos	58
5.27.3.2	setValues	59
5.28	scoreStruct Strukturreferenz	59
5.28.1	Ausführliche Beschreibung	59
5.29	Shoot Klassenreferenz	60
5.29.1	Beschreibung der Konstruktoren und Destruktoren	60
5.29.1.1	Shoot	60
5.29.2	Dokumentation der Elementfunktionen	61
5.29.2.1	getInflictedDamage	61
5.29.2.2	getOrigin	61
5.30	stateStruct Strukturreferenz	61
5.30.1	Ausführliche Beschreibung	62
6	Datei-Dokumentation	63
6.1	Wiesn-Run/src/portaudio.h-Dateireferenz	63
6.1.1	Ausführliche Beschreibung	65
6.1.2	Makro-Dokumentation	65
6.1.2.1	paClipOff	65
6.1.2.2	paCustomFormat	65
6.1.2.3	paDitherOff	66
6.1.2.4	paFloat32	66
6.1.2.5	paFormatIsSupported	66
6.1.2.6	paFramesPerBufferUnspecified	66
6.1.2.7	paInputOverflow	66
6.1.2.8	paInputUnderflow	66
6.1.2.9	paInt16	66
6.1.2.10	paInt24	67

6.1.2.11	paInt32	67
6.1.2.12	paInt8	67
6.1.2.13	paNeverDropInput	67
6.1.2.14	paNoDevice	67
6.1.2.15	paNoFlag	67
6.1.2.16	paNonInterleaved	67
6.1.2.17	paOutputOverflow	68
6.1.2.18	paOutputUnderflow	68
6.1.2.19	paPlatformSpecificFlags	68
6.1.2.20	paPrimeOutputBuffersUsingStreamCallback	68
6.1.2.21	paPrimingOutput	68
6.1.2.22	paUInt8	68
6.1.2.23	paUseHostApiSpecificDeviceSpecification	68
6.1.3	Dokumentation der benutzerdefinierten Typen	69
6.1.3.1	PaDeviceIndex	69
6.1.3.2	PaDeviceInfo	69
6.1.3.3	PaError	69
6.1.3.4	PaHostApiIndex	69
6.1.3.5	PaHostApiInfo	69
6.1.3.6	PaHostApiTypeId	69
6.1.3.7	PaHostErrorInfo	69
6.1.3.8	PaSampleFormat	70
6.1.3.9	PaStream	70
6.1.3.10	PaStreamCallback	70
6.1.3.11	PaStreamCallbackFlags	71
6.1.3.12	PaStreamCallbackResult	71
6.1.3.13	PaStreamCallbackTimeInfo	72
6.1.3.14	PaStreamFinishedCallback	72
6.1.3.15	PaStreamFlags	72
6.1.3.16	PaStreamInfo	72
6.1.3.17	PaStreamParameters	72
6.1.3.18	PaTime	73
6.1.4	Dokumentation der Aufzählungstypen	73
6.1.4.1	PaHostApiTypeId	73
6.1.4.2	PaStreamCallbackResult	73
6.1.5	Dokumentation der Funktionen	73
6.1.5.1	Pa_AbortStream	73
6.1.5.2	Pa_CloseStream	73
6.1.5.3	Pa_GetDefaultHostApi	73
6.1.5.4	Pa_GetDefaultInputDevice	74

6.1.5.5	Pa_GetDefaultOutputDevice	74
6.1.5.6	Pa_GetDeviceCount	74
6.1.5.7	Pa_GetDeviceInfo	74
6.1.5.8	Pa_GetErrorText	75
6.1.5.9	Pa_GetHostApiCount	75
6.1.5.10	Pa_GetHostApiInfo	75
6.1.5.11	Pa_GetLastHostErrorInfo	75
6.1.5.12	Pa_GetSampleSize	76
6.1.5.13	Pa_GetStreamCpuLoad	76
6.1.5.14	Pa_GetStreamHostApiType	76
6.1.5.15	Pa_GetStreamInfo	76
6.1.5.16	Pa_GetStreamReadAvailable	77
6.1.5.17	Pa_GetStreamTime	77
6.1.5.18	Pa_GetStreamWriteAvailable	77
6.1.5.19	Pa_GetVersion	77
6.1.5.20	Pa_GetVersionText	77
6.1.5.21	Pa_HostApiDeviceIndexToDeviceIndex	77
6.1.5.22	Pa_HostApiTypeIdToHostApiIndex	78
6.1.5.23	Pa_Initialize	78
6.1.5.24	Pa_IsFormatSupported	78
6.1.5.25	Pa_IsStreamActive	79
6.1.5.26	Pa_IsStreamStopped	79
6.1.5.27	Pa_OpenDefaultStream	79
6.1.5.28	Pa_OpenStream	80
6.1.5.29	Pa_ReadStream	81
6.1.5.30	Pa_SetStreamFinishedCallback	81
6.1.5.31	Pa_Sleep	82
6.1.5.32	Pa_StartStream	82
6.1.5.33	Pa_StopStream	82
6.1.5.34	Pa_Terminate	82
6.1.5.35	Pa_WriteStream	82

Kapitel 1

Ausstehende Aufgaben

Element `Enemy::Enemy` (int posX, int posY, int speedX, objectType enemy)

Skalieren der Werte und fireCooldown erhöhen

Element `Game::step` ()

Erfolgreich Schriftzug einfügen

GameOver schriftzug einfügen

Element `Player::decreaseAlcoholLevel` (int decreaseLevel)

Überflüssig, da nie aufgerufen. Auch wenn der Name es nicht vermuten lässt: increaseAlcoholLevel kann den Level auch verringern und wird benutzt.

Klasse `scoreStruct`

Das Konzept der Alkohol-Punkte muss noch ausgearbeitet werden.

Autor

Simon

Klasse `stateStruct`

Diese Struktur ist vermutlich überflüssig.

Autor

Simon

Kapitel 2

Hierarchie-Verzeichnis

2.1 Klassenhierarchie

Die Liste der Ableitungen ist -mit Einschränkungen- alphabetisch sortiert:

Audio	9
AudioControl	12
audioCooldownstruct	16
audioCooldownStruct	16
audioDistanceStruct	17
audioStruct	18
collisionStruct	18
compareGameObjects	18
compareScores	19
Input	33
Menu	37
Menu::menuEntry	41
PaDeviceInfo	43
PaHostApiInfo	44
PaHostErrorInfo	45
PaStreamCallbackTimeInfo	45
PaStreamInfo	46
PaStreamParameters	47
AudioControl::playStruct	52
QGraphicsPixmapItem	
GameObject	31
MovingObject	42
Enemy	19
Player	48
Shoot	60
PowerUp	53
QObject	
Game	22
RenderBackground	55
RenderGUI	57
scoreStruct	59
stateStruct	61

Kapitel 3

Klassen-Verzeichnis

3.1 Auflistung der Klassen

Hier folgt die Aufzählung aller Klassen, Strukturen, Varianten und Schnittstellen mit einer Kurzbeschreibung:

Audio	Die Audio-Klasse erzeugt Audioobjekte. Für jeden Audioobjekt Typ mit Name <code>type_name</code> wird zu Beginn eine Instanz der Klasse erstellt und in AudioControl an die Liste <code>audioobjects</code> angehängt. Jedes Audioobjekt liest die zum Objekt Typ gehörigen Audiosamples aus einer WAVE Datei ein und übergibt das Sample an Position <code>pos</code> per Aufruf mit getSample(int pos) an den Aufrufer . . .	9
AudioControl	Die AudioControl-Klasse synchronisiert alle Audioausgabeanweisungen und spielt passende Audioobjekte ab. Eine Instanz dieser Klasse wird innerhalb der game.h angelegt	12
audioCooldownstruct		16
audioCooldownStruct		16
audioDistanceStruct		17
audioStruct		18
collisionStruct	Struktur für die Events Enthält <code>affectedObject</code> als Objekt, aus dessen Sicht die Kollision berechnet wurde. <code>affectedObject</code> ist immer ein MovingObject , <code>causingObject</code> kann beides sein. Die Art und Richtung der Kollision werden mit gespeichert	18
compareGameObjects	Vergleich zweier <code>GameObjects</code> bezüglich der X-Position Die Methode <code>std::list::sort</code> benötigt ein struct mit einem boolschen Operator zur Sortierung. Diese Implementierung des Operators sortiert aufsteigend	18
compareScores	Vergleich zweier Scores Der Vergleich findet über die Summe der Punkte in den einzelnen Kategorien statt. Der Operator im struct ist mit größer (<code>></code>) programmiert, da die Liste absteigend sortiert werden soll	19
Enemy		19
Game	Game-Klasse Die Game-Klasse bündelt alle Kern-Funktionalitäten des Spiels. Innerhalb der <code>main.cpp</code> wird eine Instanz dieser Klasse angelegt, aus der heraus das gesamte Spiel läuft. Die einzelnen Methoden werden in der <code>game.cpp</code> jeweils erklärt	22
GameObject		31
Input	Die Input-Klasse aktualisiert die für das Spiel relevanten Tastatureingaben. Eine Instanz dieser Klasse wird innerhalb der game.h angelegt	33
Menu	Menü-Klasse eine Instanz repräsentiert ein Menü mit diesen Funktionen:	37
Menu::menuEntry	Struct zur Beschreibung eines Menü-Eintrags	41
MovingObject		42

PaDeviceInfo	43
PaHostApiInfo	44
PaHostErrorInfo	45
PaStreamCallbackTimeInfo	45
PaStreamInfo	46
PaStreamParameters	47
Player	48
AudioControl::playStruct	52
PowerUp	
Klasse für Power-Ups	53
RenderBackground	
Hintergrund-Klasse Eine Instanz wird bei jedem Levelstart in der Funktion Game::startNewGame angelegt. Die Klasse initialisiert alle Hintergrundgrafiken und aktualisiert deren Positionen im laufendem Spiel. Auch die Bewegungsparallaxe wird hier berechnet. Jede Hintergrundebene besteht immer aus zwei nebeneinander stehenden Bildern. Ist eines davon, bedingt durch die Vorwärtsbewegung des Spielers nicht mehr sichtbar, so wird es wieder am zweiten Bild vorbei, nach vorne geschoben. So wird gewährleistet das der Spieler nicht an den Bildern "vorbeiläuft"	55
RenderGUI	
Anzeigen der Spielerwerte-Klasse Eine Instanz wird bei jedem Levelstart in der Funktion Game::startNewGame angelegt. Die Klasse initialisiert alle Grafikelemente die mit der Anzeige von Spielerwerten zu tun hat (Gesundheit, Alkoholpegel, Munitionsvorrat, Punkte). Außerdem werden hier auch die angezeigten Werte im Spiel fortlaufend aktualisiert. Alle Elemente sind "Kinder" der Gesundheitsanzeige um Positionsaktualisierungen zu vereinfachen (Kindelemente verhalten sich immer relativ um Elternobjekt und werden auch automatisch mit diesem der Scene hinzugefügt bzw. auch wieder entfernt)	57
scoreStruct	
Struktur für die Score des Spielers In dieser Struktur werden Name des Spielers, getötete Gegner, zurückgelegte Entfernung und Alkohol-Punkte gespeichert. Alkohol-Punkte erhält der Spieler für einen gewissen Pegel in einem Zeitabschnitt	59
Shoot	60
stateStruct	
Struktur für die States des Spiels Sowohl Sound- als auch Grafik-Ausgabe erhalten aus den States Informationen darüber, was gerade im Spiel passiert, z.B. dass gerade der Spieler angreift, ein Gegner stirbt etc	61

Kapitel 4

Datei-Verzeichnis

4.1 Auflistung der Dateien

Hier folgt die Aufzählung aller dokumentierten Dateien mit einer Kurzbeschreibung:

Wiesn-Run/src/ audio.h	??
Wiesn-Run/src/ audiocontrol.h	??
Wiesn-Run/src/ definitions.h	??
Wiesn-Run/src/ enemy.h	??
Wiesn-Run/src/ game.h	??
Wiesn-Run/src/ gameobject.h	??
Wiesn-Run/src/ input.h	??
Wiesn-Run/src/ menu.h	??
Wiesn-Run/src/ movingobject.h	??
Wiesn-Run/src/ player.h	??
Wiesn-Run/src/ portaudio.h	
The portable PortAudio API	63
Wiesn-Run/src/ powerup.h	??
Wiesn-Run/src/ renderbackground.h	??
Wiesn-Run/src/ renderGUI.h	??
Wiesn-Run/src/ shoot.h	??

Kapitel 5

Klassen-Dokumentation

5.1 Audio Klassenreferenz

Die Audio-Klasse erzeugt Audioobjekte. Für jeden Audioobjekt Typ mit Name `type_name` wird zu Beginn eine Instanz der Klasse erstellt und in `AudioControl` an die Liste `audioobjects` angehängt. Jedes Audioobjekt liest die zum Objekt Typ gehörigen Audiosamples aus einer WAVE Datei ein und übergibt das Sample an Position `pos` per Aufruf mit `getSample(int pos)` an den Aufrufer.

```
#include <audio.h>
```

Öffentliche Methoden

- `Audio (std::string type_name)`
Konstruktor instanziiert ein Objekt der Klasse `Audio`.
- `~Audio ()`
Destruktor löscht ein Objekt der Klasse `Audio`.
- `std::string getSource ()`
`getSource` gibt bei Aufruf den Namen des Objektes zurück welcher dem Pfad in der Ressourcendatenbank entspricht.
- `float getSample (int pos)`
`getSample` gibt bei Aufruf das Sample an Position = `pos` der zu Audioobjekt gehörigen Wave Datei mit Bittiefe 16 bit zurück.
- `int getSamplenumber ()`
`getSamplenumber` gibt bei Aufruf die Anzahl an Samples der zu Audioobjekt gehörigen Wave Datei zurück.

Private Methoden

- `void readSamples ()`
`readSamples` liest bei Aufruf alle Samples der zu Audioobjekt gehörigen Wave Datei in die Variable "samples" ein. Eingelesen werden sollen RIFF Mono Wave Dateien mit 44100Hz Samplerate. Die Bittiefe ist hierbei variabel 8 oder 16bit. Es greift hierfür auf die zum Objekt gehörige, in der Ressourcendatenbank gespeicherte Wave Datei mit Pfadnamen "source" zurück. Die Funktion wertet den fmt Header des Wave File aus und liest im Anschluss den data Chunk ein. Die Bittiefe wird in float konvertiert um eine Weiterbearbeitung der Samples ohne Dynamikverlust durchführen zu können.
- `qint16 to16bitSample (quint8 sample8bit)`
`to16bitSample` konvertiert einen 8 bit integer Sample in einen 16 bit Integer Sample. Ziel ist eine einheitlich Bearbeitung der Samples verschiedener Audioobjekte vornehmen zu können.
- `void normalize ()`
`normalize` normalisiert den 16 bit Integer QVector samples. Es wird hierfür die größte Betrag-Amplitude eines Sample in samples bestimmt. Diese Amplitude wird auf den maximalen signed Integer 16 Bit Wert gesetzt. Alle anderen Samples werden entsprechend ihres Verhältnisses zur größten Betrag-Amplitude skaliert.

Private Attribute

- `std::string source`
source speichert den Namen des Audioobjekts als string welcher dem Dateinamen der zugehörigen Wave Datei entspricht.
- `std::vector< float > samples`
samples speichert die normalisierten samples des [Audio](#) Objekts als QVektor mit 32 bit float Werten.
- `int samplenumber`
samplenumber speichert die Anzahl an Samples in der gesamten [Audio](#) Datei des [Audio](#) Objekts als Integer.

5.1.1 Ausführliche Beschreibung

Die Audio-Klasse erzeugt Audioobjekte. Für jeden Audioobjekt Typ mit Name `type_name` wird zu Beginn eine Instanz der Klasse erstellt und in [AudioControl](#) an die Liste `audioobjects` angehängt. Jedes Audioobjekt liest die zum Objekt Typ gehörigen Audiosamples aus einer WAVE Datei ein und übergibt das Sample an Position `pos` per Aufruf mit [getSample\(int pos\)](#) an den Aufrufer.

Autor

Felix Pfreundtner

5.1.2 Beschreibung der Konstruktoren und Destruktoren

5.1.2.1 `Audio::Audio (std::string type_name)`

Konstruktor instanziert ein Objekt der Klasse [Audio](#).

Autor

Felix Pfreundtner

5.1.2.2 `Audio::~~Audio ()`

Destruktor löscht ein Objekt der Klasse [Audio](#).

Autor

Felix Pfreundtner

5.1.3 Dokumentation der Elementfunktionen

5.1.3.1 `float Audio::getSample (int pos)`

`getSample` gibt bei Aufruf das Sample an Position = `pos` der zu Audioobjekt gehörigen Wave Datei mit Bittiefe 16 bit zurück.

Rückgabe

`float sample`

Autor

Felix Pfreundtner

5.1.3.2 `int Audio::getSplenumber ()`

`getSplenumber` gibt bei Aufruf die Anzahl an Samples der zu Audioobjekt gehörigen Wave Datei zurück.

Rückgabe

`int` Instanzvariable `splenumber`

Autor

Felix Pfreundtner

5.1.3.3 `std::string Audio::getSource ()`

`getSource` gibt bei Aufruf den Namen des Objektes zurück welcher dem Pfad in der Ressourcendatenbank entspricht.

Rückgabe

`std::string` `source`

Autor

Felix Pfreundtner

5.1.3.4 `void Audio::normalize () [private]`

`normalize` normalisiert den 16 bit Integer QVector samples. Es wird hierfür die größte Betrag-Amplitude eines Sample in samples bestimmt. Diese Amplitude wird auf den maximalen signed Integer 16 Bit Wert gesetzt. Alle anderen Samples werden entsprechend ihres Verhältnisses zur größten Betrag-Amplitude skaliert.

Autor

Felix Pfreundtner

5.1.3.5 `void Audio::readSamples () [private]`

`readSamples` liest bei Aufruf alle Samples der zu Audioobjekt gehörigen Wave Datei in die Variable "samples" ein. Eingelesen werden sollen RIFF Mono Wave Dateien mit 44100Hz Samplerate. Die Bittiefe ist hierbei variabel 8 oder 16bit. Es greift hierfür auf die zum Objekt gehörige, in der Ressourcendatenbank gespeicherte Wave Datei mit Pfadnamen "source" zurück. Die Funktion wertet den fmt Header des Wave File aus und liest im Anschluss den data Chunk ein. Die Bittiefe wird in float konvertiert um eine Weiterbearbeitung der Samples ohne Dynamikverlust durchführen zu können.

Autor

Felix Pfreundtner

lese den Namen des Headers des nächsten Chunks aus

5.1.3.6 `qint16 Audio::to16bitSample (quint8 sample8bit) [private]`

`to16bitSample` konvertiert einen 8 bit integer Sample in einen 16 bit Integer Sample. Ziel ist eine einheitlich Bearbeitung der Samples verschiedener Audioobjekte vornehmen zu können.

Parameter

<code>quint8</code>	<code>sample8bit</code>
---------------------	-------------------------

Rückgabe

`quint16 sample16bit`

Autor

Felix Pfreundtner

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- `Wiesn-Run/src/audio.h`
- `Wiesn-Run/src/audio.cpp`

5.2 AudioControl Klassenreferenz

Die AudioControl-Klasse synchronisiert alle Audioausgabeanweisungen und spielt passende Audioobjekte ab. Eine Instanz dieser Klasse wird innerhalb der [game.h](#) angelegt.

```
#include <audiocontrol.h>
```

Klassen

- struct [playStruct](#)

Öffentliche Methoden

- [AudioControl](#) ()
Konstruktor instanziert ein Objekt der Klasse [AudioControl](#).
- [~AudioControl](#) ()
Destruktor löscht ein Objekt der Klasse [AudioControl](#).
- void [playInitialize](#) ()
playInitialize initialisiert die Abspielbibliothek Portaudio, öffnet den PortAudio Stream pastream und startet eine Call-back Audiowiedergabe
- void [playTerminate](#) ()
playTerminate stoppt bei Aufruf die PortAudio Audioausgabe, beendet im Anschluss den Portaudio Stream und beendet zuletzt PortAudio.
- void [updatePlayevents](#) (std::list< struct [audioStruct](#) > *audioevents)
updatePlayevents aktualisiert nach Aufruf über [Game::step](#) alle im Moment abgespielten, in der Liste "playevents" gespeicherten [playStruct](#)'s mit aktuellen [audioStruct](#)'s aus der übergebenen Liste audioevents.

Private Typen

- enum [statusFilter](#) { `no`, `alcohol`, `lifecritical` }
statusFilter definiert alle [Audio](#) Filter Status Optionen

Private Methoden

- int [instancepaCallback](#) (const void *input, void *output, unsigned long frameCount, const [PaStreamCallbackTimeInfo](#) *timeInfo, [PaStreamCallbackFlags](#) statusFlags)

instancepaCallback wird von Portaudio aufgerufen wenn nahezu letzter Audioblock abgespielt wurde und neu Audio-samples benötigt werden.

Private, statische Methoden

- static int [staticpaCallback](#) (const void *input, void *output, unsigned long frameCount, const [PaStreamCallbackTimeInfo](#) *timeInfo, [PaStreamCallbackFlags](#) statusFlags, void *userData)

staticpaCallback ist die Statische Callback Funktion der [AudioControl](#) Klasse. Die Funktion wird immer dann aufgerufen, wenn der PortAudio Stream einen neuen Ausgabeblock benötigt, da der letzte abgespielt wurde. Die Funktion ruft die Funktion *instancepaCallback* auf, welche nicht statisch ist und auf alle instance variablen und Funktionen (des von [Game](#) erzeugten [AudioControl](#) Objektes *audioOutput*) zugreifen kann. Dies ermöglicht einen Einfachen Austausch von [Audio](#) Blöcken zwischen [Game](#) Thread und Portaudio Wiedergabethread.

Private Attribute

- std::mutex [mtx](#)

mtx ist eine Mutex, welche zwischen dem [Game](#) Thread und dem PortAudio Ausgabe Thread vermittelt. Es muss die gleichzeitig von [Game](#) über [updatePlayevents\(\)](#) beschriebene und PortAudio über [instancepaCallback\(\)](#) gelesene Liste *playevents* gelockt werden.

- std::list< [playStruct](#) > [playevents](#)

playevents beinhaltet eine Liste mit allen im Moment abgespielten *playStructs*.

- std::vector< [Audio](#) > [audioobjects](#)

audioobjects beinhaltet eine Array mit allen vorhandenen Objekten der Klasse [Audio](#)(*beispielsweise deren Samples als QVector*).

- int [waitinms](#)

waitinms speichert die wartezeit bis zum Beenden von PortAudio in Millisekunden.

- [PaError](#) [playinitializeerror](#)

playinitializeerror speichert eventuell auftretende Error beim Öffnen und Schließen des PortAudio Streams.

- int [max_playevents](#)

max_playevents definiert die maximale Anzahl an abgespielten *playEvents* ohne Clipping Effekte.

- int [blockcounter](#)

blockcounter zählt die bereits abgespielten [Audio](#) Ausgabe Blöcke.

- float [mixed_sample](#)

mixed_sample beinhaltet das aktuell von *mixSample()* gemixte Sample aller *audioEvents*.

- int [playeventsnumber](#)

playeventsnumber beinhaltet die Anzahl an aktuelle abzuspielenden *audioEvents*. Float Format da mit diesem Wert in *mixsamples* effizient gerechnet werden muss ohne Castumwandlung Integer in Float.

- [PaStream](#) * [pastream](#)

pastream ist ein Zeiger auf den PortAudio Stream.

- [PaError](#) [paerror](#)

paerror speichert einen eventuellen PortAudio Error.

- int [status_filter](#)

status_filter gibt den Filterstatus an. Wenn kein Audioevent in der *audiovents* List den Type *status_alcohol* hat -> enum none-> 0. Wenn mindestens ein Audioevent in der *audiovents* List den Type *status_alcohol* hat -> enum alcohol-> 1. Wenn mindestens ein Audioevent in der *audiovents* List den Type *status_life* hat -> enum alcohol-> 2. Wenn mindestens ein Audioevent in der *audiovents* List den Type *status_lifecritical* hat -> enum alcohol-> 3.

5.2.1 Ausführliche Beschreibung

Die AudioControl-Klasse synchronisiert alle Audioausgabeanweisungen und spielt passende Audioobjekte ab. Eine Instanz dieser Klasse wird innerhalb der [game.h](#) angelegt.

Autor

Felix Pfreundtner

5.2.2 Beschreibung der Konstruktoren und Destruktoren

5.2.2.1 AudioControl::AudioControl ()

Konstruktor instanziert ein Objekt der Klasse [AudioControl](#).

Autor

Felix Pfreundtner

source: <http://www.freesound.org/people/qubodup/sounds/169725/>

5.2.2.2 AudioControl::~~AudioControl ()

Destruktor löscht ein Objekt der Klasse [AudioControl](#).

Autor

Felix Pfreundtner

5.2.3 Dokumentation der Elementfunktionen

5.2.3.1 `int AudioControl::instancePaCallback (const void * inputBuffer, void * outputBuffer, unsigned long framesPerBuffer, const PaStreamCallbackTimeInfo * timeInfo, PaStreamCallbackFlags statusFlags) [private]`

`instancePaCallback` wird von Portaudio aufgerufen wenn nahezu letzter Audioblock abgespielt wurde und neu Audiosamples benötigt werden.

Parameter

<i>const</i>	<code>void *inputBuffer</code>
<i>void</i>	<code>*outputBuffer</code>
<i>unsigned</i>	<code>long framesPerBuffer,</code>
<i>const</i>	<code>PaStreamCallbackTimeInfo* timeInfo,</code>
<i>PaStream-CallbackFlags</i>	<code>statusFlags</code>

Rückgabe

`int` returncode

Autor

Felix Pfreundtner

5.2.3.2 void AudioControl::playInitialize ()

playInitialize initialisiert die Abspielbibliothek Portaudio, öffnet den PortAudio Stream pastream und startet eine Callback Audiowiedergabe

Autor

Felix Pfreundtner

5.2.3.3 void AudioControl::playTerminate ()

playTerminate stoppt bei Aufruf die PortAudio Audioausgabe, beendet im Anschluss den Portaudio Stream und beendet zuletzt PortAudio.

Autor

Felix Pfreundtner

5.2.3.4 static int AudioControl::staticpaCallback (const void * *input*, void * *output*, unsigned long *frameCount*, const PaStreamCallbackTimeInfo * *timeInfo*, PaStreamCallbackFlags *statusFlags*, void * *userData*)
[inline],[static],[private]

staticpaCallback ist die Statische Callback Funktion der [AudioControl](#) Klasse. Die Funktion wird immer dann aufgerufen, wenn der PortAudio Stream einen neuen Ausgabeblock benötigt, da der letzte abgespielt wurde. Die Funktion ruft die Funktion instancepaCallback auf, welche nicht statisch ist und auf alle instance variablen und Funktionen (des von [Game](#) erzeugten [AudioControl](#) Objektes audioOutput) zugreifen kann. Dies ermöglicht einen Einfachen Austausch von [Audio](#) Blöcken zwischen [Game](#) Thread und Portaudio Wiedergabethread.

Parameter

<i>const</i>	void *inputBuffer
<i>void</i>	*outputBuffer
<i>unsigned</i>	long framesPerBuffer,
<i>const</i>	PaStreamCallbackTimeInfo* timeInfo,
<i>PaStream-CallbackFlags</i>	statusFlags

Rückgabe

((AudioControl*)userData) ->instancepaCallback(input, output, frameCount, timeInfo, statusFlags)

Autor

Felix Pfreundtner

5.2.3.5 void AudioControl::updatePlayevents (std::list< struct audioStruct > * *audioevents*)

updatePlayevents aktualisiert nach Aufruf über [Game::step](#) alle im Moment abgespielten, in der Liste "playevents" gespeicherten [playStruct](#)'s mit aktuellen [audioStruct](#)'s aus der übergebenen Liste audioevents.

Parameter

<code>std::list<struct</code>	<code>audioStruct> *audioevents</code>
----------------------------------	---

Autor

Felix Pfreundtner

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- Wiesn-Run/src/audiocontrol.h
- Wiesn-Run/src/audiocontrol.cpp

5.3 audioCooldownstruct Strukturreferenz

Öffentliche Attribute

- struct `audioStruct` **audioEvent**
- `std::chrono::duration< int,`
`std::milli >` **cooldown**

Die Dokumentation für diese Struktur wurde erzeugt aufgrund der Datei:

- Wiesn-Run/src/definitions.h

5.4 audioCooldownStruct Strukturreferenz

Öffentliche Attribute

- `std::chrono::duration< int,`
`std::milli >` **scene_flyingbeer** = `std::chrono::milliseconds(0)`
- `std::chrono::duration< int,`
`std::milli >` **scene_enemy_security** = `std::chrono::milliseconds(0)`
- `std::chrono::duration< int,`
`std::milli >` **scene_enemy_tourist** = `std::chrono::milliseconds(0)`
- `std::chrono::duration< int,`
`std::milli >` **scene_enemy_boss** = `std::chrono::milliseconds(0)`
- `std::chrono::duration< int,`
`std::milli >` **scene_collision_obstacle** = `std::chrono::milliseconds(500)`
- `std::chrono::duration< int,`
`std::milli >` **scene_collision_enemy** = `std::chrono::milliseconds(1639)`
- `std::chrono::duration< int,`
`std::milli >` **scene_collision_player** = `std::chrono::milliseconds(899)`
- `std::chrono::duration< int,`
`std::milli >` **scene_collision_flyingbeer** = `std::chrono::milliseconds(1211)`
- `std::chrono::duration< int,`
`std::milli >` **powerup_beer** = `std::chrono::milliseconds(2989)`
- `std::chrono::duration< int,`
`std::milli >` **powerup_food** = `std::chrono::milliseconds(3989)`
- `std::chrono::duration< int,`
`std::milli >` **status_alcohol** = `std::chrono::milliseconds(0)`
- `std::chrono::duration< int,`
`std::milli >` **status_life** = `std::chrono::milliseconds(0)`
- `std::chrono::duration< int,`
`std::milli >` **status_lifecritical** = `std::chrono::milliseconds(0)`

- `std::chrono::duration< int,`
 `std::milli > status_dead = std::chrono::milliseconds(4989)`
- `std::chrono::duration< int,`
 `std::milli > player_walk = std::chrono::milliseconds(0)`
- `std::chrono::duration< int,`
 `std::milli > player_jump = std::chrono::milliseconds(700)`
- `std::chrono::duration< int,`
 `std::milli > background_menu = std::chrono::milliseconds(0)`
- `std::chrono::duration< int,`
 `std::milli > background_highscore = std::chrono::milliseconds(0)`
- `std::chrono::duration< int,`
 `std::milli > background_level1 = std::chrono::milliseconds(0)`
- `std::chrono::duration< int,`
 `std::milli > background_level2 = std::chrono::milliseconds(0)`
- `std::chrono::duration< int,`
 `std::milli > background_level3 = std::chrono::milliseconds(0)`
- `std::chrono::duration< int,`
 `std::milli > background_startgame = std::chrono::milliseconds(3000)`
- `std::chrono::duration< int,`
 `std::milli > background_levelfinished = std::chrono::milliseconds(5365)`

Die Dokumentation für diese Struktur wurde erzeugt aufgrund der Datei:

- `Wiesn-Run/src/definitions.h`

5.5 audioDistanceStruct Strukturreferenz

Öffentliche Attribute

- `float scene_flyingbeer = -1`
- `float scene_enemy_tourist = -1`
- `float scene_enemy_security = -1`
- `float scene_enemy_boss = -1`
- `float scene_collision_obstacle = 0`
- `float scene_collision_enemy = 0`
- `float scene_collision_player = 0`
- `float scene_collision_flyingbeer = 0`
- `float powerup_beer = 0`
- `float powerup_food = 0`
- `float status_alcohol = 0`
- `float status_life = 0`
- `float status_lifecritical = 0`
- `float status_dead = 0`
- `float player_walk = 0`
- `float player_jump = 0`
- `float background_menu = 0.0`
- `float background_highscore = 0.1`
- `float background_level1 = 0.3`
- `float background_level2 = 0.3`
- `float background_level3 = 0.3`
- `float background_startgame = 0.5`
- `float background_levelfinished = 0.0`

Die Dokumentation für diese Struktur wurde erzeugt aufgrund der Datei:

- `Wiesn-Run/src/definitions.h`

5.6 audioStruct Strukturreferenz

Öffentliche Attribute

- int **id**
- audioType **type**
- float **distance**

Die Dokumentation für diese Struktur wurde erzeugt aufgrund der Datei:

- Wiesn-Run/src/definitions.h

5.7 collisionStruct Strukturreferenz

Struktur für die Events Enthält affectedObject als Objekt, aus dessen Sicht die Kollision berechnet wurde. affected-Object ist immer ein [MovingObject](#), causingObject kann beides sein. Die Art und Richtung der Kollision werden mit gespeichert.

```
#include <game.h>
```

Öffentliche Attribute

- [GameObject](#) * **affectedObject**
- [GameObject](#) * **causingObject**
- enum collisionDirection **direction**

5.7.1 Ausführliche Beschreibung

Struktur für die Events Enthält affectedObject als Objekt, aus dessen Sicht die Kollision berechnet wurde. affected-Object ist immer ein [MovingObject](#), causingObject kann beides sein. Die Art und Richtung der Kollision werden mit gespeichert.

Autor

Simon, Johann(15.6)

Die Dokumentation für diese Struktur wurde erzeugt aufgrund der Datei:

- Wiesn-Run/src/game.h

5.8 compareGameObjects Strukturreferenz

Vergleich zweier GameObjects bezüglich der X-Position Die Methode std::list::sort benötigt ein struct mit einem boolschen Operator zur Sortierung. Diese Implementierung des Operators sortiert aufsteigend.

Öffentliche Methoden

- bool **operator()** ([GameObject](#) *objA, [GameObject](#) *objB)

5.8.1 Ausführliche Beschreibung

Vergleich zweier GameObjects bezüglich der X-Position Die Methode std::list::sort benötigt ein struct mit einem boolschen Operator zur Sortierung. Diese Implementierung des Operators sortiert aufsteigend.

Parameter

1.Objekt	
2.Objekt	

Rückgabe

true, wenn 1.Objekt weiter links als 2.Objekt

Autor

Simon

Die Dokumentation für diese Struktur wurde erzeugt aufgrund der Datei:

- Wiesn-Run/src/game.cpp

5.9 compareScores Strukturreferenz

Vergleich zweier Scores Der Vergleich findet über die Summe der Punkte in den einzelnen Kategorien statt. Der Operator im struct ist mit größer (>) programmiert, da die Liste absteigend sortiert werden soll.

Öffentliche Methoden

- bool **operator()** (scoreStruct scoreA, scoreStruct scoreB)

5.9.1 Ausführliche Beschreibung

Vergleich zweier Scores Der Vergleich findet über die Summe der Punkte in den einzelnen Kategorien statt. Der Operator im struct ist mit größer (>) programmiert, da die Liste absteigend sortiert werden soll.

Autor

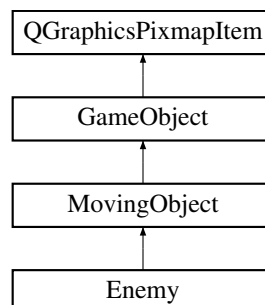
Simon

Die Dokumentation für diese Struktur wurde erzeugt aufgrund der Datei:

- Wiesn-Run/src/game.cpp

5.10 Enemy Klassenreferenz

Klassendiagramm für Enemy:



Öffentliche Methoden

- **Enemy** (int posX, int posY, int speedX, objectType enemy)
Konstruktor für ein Enemy-Objekt.
- int **getHealth** () const
Enemy::getHealth Gibt Lebensstand zurück.
- void **setHealth** (int health)
Enemy::setHealth Lebensstand wird gesetzt.
- bool **receiveDamage** (int damage)
- int **getInflictedDamage** () const
Enemy::getInflictedDamage gibt Schaden zurück, den der gegner zufügt.
- int **getFireCooldown** () const
Enemy::getFireCooldown.
- bool **getDeath** () const
Enemy::getDeath Gibt an ob der Gegner Tot ist.
- void **setDeath** (bool death)
Enemy::setDeath Zustand-TOT wird gesetzt.
- int **getDeathCooldown** () const
Enemy::getDeathCooldown.
- virtual void **update** ()
Enemy::update führt Bewegungen des Gegners aus.

Private Attribute

- int **health**
- int **fireRate**
- int **fireCooldown**
- int **inflictedDamage**
- bool **death**
- int **DeathCooldown**

Weitere Geerbte Elemente

5.10.1 Beschreibung der Konstruktoren und Destruktoren

5.10.1.1 Enemy::Enemy (int posX, int posY, int speedX, objectType enemy)

Konstruktor für ein Enemy-Objekt.

Class **Enemy** lastUpdate: **update()** 10.6 Johann

Parameter

<i>posX</i>	: X-Position
<i>posY</i>	: Y-Position
<i>speedX</i>	: Geschwindigkeit in X-Richtung

Noch zu erledigen Skalieren der Werte und fireCooldown erhöhen

5.10.2 Dokumentation der Elementfunktionen

5.10.2.1 `bool Enemy::getDeath () const`

[Enemy::getDeath](#) Gibt an ob der Gegner Tot ist.

Rückgabe

: Zustand - TOT

5.10.2.2 `int Enemy::getDeathCooldown () const`

[Enemy::getDeathCooldown](#).

Rückgabe

deathCooldown

5.10.2.3 `int Enemy::getFireCooldown () const`

[Enemy::getFireCooldown](#).

Rückgabe

fireCooldown

5.10.2.4 `int Enemy::getHealth () const`

[Enemy::getHealth](#) Gibt Lebensstand zurück.

Rückgabe

: Lebensstand

5.10.2.5 `int Enemy::getInflictedDamage () const`

[Enemy::getInflictedDamage](#) gibt Schaden zurück, den der gegner zufügt.

Rückgabe

: Schaden

5.10.2.6 `void Enemy::setDeath (bool death)`

[Enemy::setDeath](#) Zustand-TOT wird gesetzt.

Parameter

<code>death</code>	: Zustand-TOT
--------------------	---------------

5.10.2.7 void Enemy::setHealth (int health)

[Enemy::setHealth](#) Lebensstand wird gesetzt.

Parameter

<code>health</code>	: Lebensstand
---------------------	---------------

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

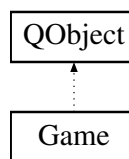
- Wiesn-Run/src/enemy.h
- Wiesn-Run/src/enemy.cpp

5.11 Game Klassenreferenz

Game-Klasse Die Game-Klasse bündelt alle Kern-Funktionalitäten des Spiels. Innerhalb der main.cpp wird eine Instanz dieser Klasse angelegt, aus der heraus das gesamte Spiel läuft. Die einzelnen Methoden werden in der game.cpp jeweils erklärt.

```
#include <game.h>
```

Klassendiagramm für Game:



Öffentliche Methoden

- [Game](#) (int argc, char *argv[])
Konstruktor und Destruktor.
- int [step](#) ()
Game-Loop Diese Funktion wird von [timerEvent\(\)](#) aufgerufen und ist für den kompletten Ablauf des Spiels verantwortlich. grober Ablauf: LOOP:
- int [run](#) (QApplication &app)
- int [start](#) ()
Starten der Applikation.
- void [setState](#) (enum gameState newState)
Hilfsfunktion.

Öffentliche Attribute

- struct [stateStruct](#) **gameStats**
- std::list< struct [collisionStruct](#) > [collisionsToHandle](#)
Liste von Kollisionen.

Geschützte Methoden

- void [timerEvent](#) (QTimerEvent *event)

wird regelmäßig aufgerufen event muss drinstehen, damit der Timer die Funktion aufruft

Private Typen

- enum [menuIds](#) {
[menuId_NonClickable](#), [menuStartId_NewGame](#), [menuStartId_EndGame](#), [menuStartId_Help](#),
[menuStartId_Credits](#), [menuCreditsId_Back](#), [menuLevelId_Back](#), [menuLevelId_Demo](#),
[menuLevelId_Level1](#), [menuLevelId_Level2](#), [menuLevelId_Level3](#), [menuLevelId_StartGame](#),
[menuBreakId_Resume](#), [menuBreakId_EarlyEnd](#), [menuBreakId_EndGame](#), [menuStatisticsId_Next](#),
[menuNameId_Next](#), [menuHighscoreId_Next](#), [menuHelpId_Back](#) }

zur Unterscheidung und Identifizierung der Menü-Einträge

Private Methoden

- void [startNewGame](#) (QString levelFileName, int levelNum)

Level starten und beenden.

- void [loadLevelFile](#) (QString fileSpecifiler)

Level-Datei auslesen.

- void [updateHighScore](#) (std::string mode)

[Game::updateHighScore](#) Diese Funktion liest und aktualisiert die Highscore des Spiels. Als Parameter wird ein std::string mode erwartet. Ist der mode = "write", so wird die aktuelle Highscore unter Berücksichtigung der aktuellen playerScore neu geschrieben. Alle anderen Werte für mode lesen nur die alte Highscore und die des Spielers in die Liste ein, um sie z.B. im Highscore-Menü anzuzeigen. Dazu wird versucht, die Datei "wiesnHighscore.txt" auszulesen. Ist dies nicht möglich, so wurde das Spiel in dem aktuellen Verzeichnis noch nie gestartet. Falls die Datei gefunden und gelesen werden kann, so wird jeder Highscore-Eintrag in die scoreList aufgenommen. Anschließend wird die Liste nach der Summe der Punkte absteigend sortiert, und nur die 10 besten Elemente werden gespeichert. Wurde für das aktuelle Spiel eine Score angelegt und in der scoreList gespeichert, so wird dieser Eintrag eingeordnet und gegebenenfalls auch abgespeichert.

- void [displayStatistics](#) ()

füllt das Statistik- und HighscoreMenü löscht das Statistik- und Highscore-Menü und füllt es mit aktuellen Werten

- void [endGame](#) ()

[Game::endGame](#) Diese Funktion löscht nicht mehr nötige Variablen und Objekte wenn vom Spiel in das Statistikmenü gewechselt wird. es werden auch die Statistik und Highscoremenüs aktualisiert.

- void [appendWorldObjects](#) (Player *playerPointer)

Funktionen in der Loop.

- void [reduceWorldObjects](#) (Player *playerPointer)

[Game::reduceWorldObjects](#).

- void [evaluateInput](#) ()

Checkt welche Tasten für die Spielkontrolle gedrückt sind mögliche Tasten:

- void [calculateMovement](#) ()

Geht die worldObjects durch und aktualisiert bei jedem die Position, Gegner bei denen der DeathCooldown abgelaufen ist, werden zum löschen vorgemerkt, Gegner bei denen der FireCooldown abgelaufen ist feuern. wird momentan auch über Debug ausgegeben.

- void [detectCollision](#) (std::list< [GameObject](#) * > *objectsToCalculate)

[Game::detectCollision](#) Diese Funktion berechnet, ob Kollisionen zwischen benachbarten Objekten auftreten und falls ja, aus welcher Richtung diese stattfinden. Da die Liste worldObjects in jedem Zeitschritt sortiert wird, müssen die Kollisionen nur für die nächsten Nachbarn berechnet werden. Allerdings können durch ungünstige Lage auch Objekte kollidieren, die nicht direkt nebeneinander in der Liste liegen. Dafür werden die fünf Nachbarn links und rechts jedes MovingObjects geprüft, falls vorhanden.

- void [handleCollisions](#) ()

Kollisionen in der Liste `collisionsToHandle` werden der Reihe nach aus Sicht des `affectedObjects` bearbeitet. In einer Schleife wird das jeweils erst `CollisionEvent` bearbeitet. Dabei werden nur an dem Objekt `affectedObject` Änderungen vorgenommen. Mögliche Objekte: `Spieler(player)`, `Gegner(enemy)`, `Bierkrug(shot)` mögliche Kollision mit `Spieler(player)`, `Hindernis(obstacle)`, `Gegner(enemy)`, `Bierkrug(shot)`, `Power-Up(powerUp)`

- void `updateScore` ()
`Game::updateScore` Aktualisiert die Score des Spielers. Diese Score wird von der Grafik während des Spiels ausgegeben und am Ende des Spiels in die Highscore aufgenommen.
- void `updateAudioevents` ()
durchläuft die Liste `audioStorage` und zählt die `Audio` Cooldowns herunter. Die Soundevents die noch laufen, werden an die Liste `AudioEvents` übergeben. Die fertigen werden gelöscht.
- void `renderGraphics` (std::list< `GameObject` * > *objectList, `Player` *playerPointer)
`Game::renderGraphics` Positionssaktualisierungen der Grafiken aller Beewglichen Objekte.
- void `menuInit` ()
Funktionen zu Start und Ende der Applikation.
- void `exitGame` ()
Diese Funktion wird aufgerufen wenn das Programm beendet werden soll.
- bool `eventFilter` (QObject *obj, QEvent *event)
`eventFilter` wird aufgerufen, wenn ein neues QEvent auftritt. Diese Funktion überwacht die Betätigung von Tastatur Eingaben und handelt den Aufruf des QT Schließ-Button (x) im Spielfenster. Die Tastatureingaben werden über das `keyInput` Objekt ausgewertet. Der Aufruf des QT Schließ-Button (x) ist neben dem Aufruf des Hauptmenüeintrags Exit die 2. Möglichkeit das Spiel zu beenden. Wird ein CloseEvent festgestellt wird die Variable `exitGameevent` auf False gesetzt und das Spiel zum Ende des aktuellen Steps in `Game::timerEvent` beendet.
- int `getStepIntervall` ()
Hilfsfunktion.
- void `timeNeeded` (string name)

Private Attribute

- std::list< `GameObject` * > `worldObjects`
In der Welt befindliche Objekte.
- std::list< `GameObject` * > `levelSpawn`
Statische Objekte, die zu Anfang gespawnt werden.
- std::list< `GameObject` * > `objectsToDelete`
Zu löschende Schüsse.
- `AudioControl` * `audioOutput`
Audiocontrol Objekt welches aktuelle Audioevents auswertet.
- std::thread `portaudiothread`
`Audio` Wiedergabe Thread welcher Portaudio Callback Funktion ausführt und Audioevents Blockweise abspielt.
- std::list< struct `audioStruct` > `audioevents`
Liste `audioevents` mit allen im Step stattfindenden `AudioStructs`.
- std::list< struct `audioCooldownstruct` > `audioStorage`
Liste mit den Audioevents die einmal aufgerufen werden aber eine Längere Spielzeit haben.
- int `sceneWidth`
Breite der Szene.
- int `levelLength` = 0
Länge des Levels.
- std::list< struct `scoreStruct` > `scoreList`
- struct `scoreStruct` `playerScore`
- int `stepIntervall`
Länge eines Steps.
- bool `exitGameevent`
Spiel Beenden gedrückt.

- bool `levelStartevent`
Spiel Starten gedrückt.
- `Player` * `playerObjPointer`
- `QGraphicsView` * `window`
für das Ausgabefenster QGraphicsView und QGraphicsScene der Level
- `QGraphicsScene` * `levelScene`
- `RenderGUI` * `showGUI`
für alle Anzeigen wie Leben,Alkohol,Score,..
- `RenderBackground` * `showBackground`
für die Hintergrundgrafiken
- `QApplication` * `appPointer`
Zeiger auf QApplication.
- `std::chrono::high_resolution_clock::time_point` `letzterAufruf`
für Zeitmessung
- `Input` * `keyInput` = new `Input()`
Erstelle Input Objekt zum Aufzeichnen der Keyboard Inputs.
- enum gameState `state` = gameMenuStart
- `Menu` * `aktMenu` = `menuStart`
aktueller Spielzustand
- `Menu` * `menuStart`
aktuell aktives Menü, null während das Spiel läuft; wird in setState gesetzt
- `Menu` * `menuCredits`
- `Menu` * `menuLevel`
- `Menu` * `menuBreak`
- `Menu` * `menuStatistics`
- `Menu` * `menuName`
- `Menu` * `menuHighscore`
- `Menu` * `menuHelp`
- int `stepCount` = 0
- int `audioIDs`
- `audioCooldownStruct` `audioCooldown`
- `audioDistanceStruct` `audioDistance`
- `chrono::high_resolution_clock::time_point` `thisStep`
- `chrono::high_resolution_clock::time_point` `testStep`

5.11.1 Ausführliche Beschreibung

Game-Klasse Die Game-Klasse bündelt alle Kern-Funktionalitäten des Spiels. Innerhalb der main.cpp wird eine Instanz dieser Klasse angelegt, aus der heraus das gesamte Spiel läuft. Die einzelnen Methoden werden in der game.cpp jeweils erklärt.

funtion handleCollisions hinzugefügt

Autor

Simon, Johann, Felix

5.11.2 Beschreibung der Konstruktoren und Destruktoren

5.11.2.1 Game::Game (int argc, char * argv[])

Konstruktor und Destruktor.

Konstruktor Initialisiert den appPointer.

Parameter

<i>argc</i>	
<i>argv</i>	

Autor

Rupert

5.11.3 Dokumentation der Elementfunktionen

5.11.3.1 void Game::appendWorldObjects (**Player * *playerPointer***) [private]

Funktionen in der Loop.

[Game::appendWorldObjects.](#)

Parameter

<i>playerPointer</i>	Diese Funktion fügt der Spielwelt dynamisch Gegner hinzu. In jedem Zeitschritt wird die sortierte Liste levelSpawn vom Anfang her durchlaufen. Ist die Distanz des Spielers zum Gegner kleiner als die Distanz levelSpawn, so wird das Objekt den worldObjects hinzugefügt und aus levelSpawn gelöscht. Die for-Schleife läuft solange, bis das erste Mal ein Objekt weiter als levelSpawn vom Spieler entfernt ist. Dann wird abgebrochen, da alle folgenden Objekte auf Grund der Sortierung noch weiter entfernt sein werden.
----------------------	--

Autor

Simon

5.11.3.2 void Game::calculateMovement () [private]

Geht die worldObjects durch und aktualisiert bei jedem die Position, Gegner bei denen der DeathCooldown abgelaufen ist, werden zum löschen vorgemerkt, Gegner bei denen der FireCooldown abgelaufen ist feuern. wird momentan auch über Debug ausgegeben.

Autor

Rupert, Johann

5.11.3.3 void Game::detectCollision (**std::list< GameObject * > * *objectsToCalculate***) [private]

[Game::detectCollision](#) Diese Funktion berechnet, ob Kollisionen zwischen benachbarten Objekten auftreten und falls ja, aus welcher Richtung diese stattfinden. Da die Liste worldObjects in jedem Zeitschritt sortiert wird, müssen die Kollisionen nur für die nächsten Nachbarn berechnet werden. Allerdings können durch ungünstige Lage auch Objekte kollidieren, die nicht direkt nebeneinander in der Liste liegen. Dafür werden die fünf Nachbarn links und rechts jedes MovingObjects geprüft, falls vorhanden.

Autor

Simon

5.11.3.4 void Game::displayStatistics () [private]

füllt das Statistik- und HighscoreMenü löscht das Statistik- und Highscore-Menü und füllt es mit aktuellen Werten

Autor

Rupert

5.11.3.5 void Game::endGame () [private]

[Game::endGame](#) Diese Funktion löscht nicht mehr nötige Variablen und Objekte wenn vom Spiel in das Statistikmenü gewechselt wird. es werden auch die Statistik und Highscoremenüs aktualisiert.

Autor

: Felix, Johann

5.11.3.6 void Game::evaluateInput () [private]

Checkt welche Tasten für die Spielkontrolle gedrückt sind mögliche Tasten:

- Pfeil rechts zum laufen
- Pfeil hoch zum springen
- Leertaste zum schießen
- ESC für Menü

Autor

Rupert

5.11.3.7 bool Game::eventFilter (QObject * obj, QEvent * event) [private]

eventFilter wird aufgerufen, wenn ein neues QEvent auftritt. Diese Funktion überwacht die Betätigung von Tastatur Eingaben und handelt den Aufruf des QT Schließ-Button (x) im Spielfenster. Die Tastatureingaben werden über das keyInput Objekt ausgewertet. Der Aufruf des QT Schließ-Button (x) ist neben dem Aufruf des Hauptmenüeintrags Exit die 2. Möglichkeit das Spiel zu beenden. Wird ein CloseEvent festgestellt wird die Variable exitGameevent auf False gesetzt und das Spiel zum Ende des aktuellen Steps in [Game::timerEvent](#) beendet.

Parameter

	QObject *obj
	QEvent *event

Rückgabe

: QObject::eventFilter(obj, event)

Autor

: Felix

5.11.3.8 void Game::exitGame () [private]

Diese Funktion wird aufgerufen wenn das Programm beendet werden soll.

Autor

: Felix

5.11.3.9 int Game::getStepIntervall () [private]

Hilfsfunktion.

gibt stepIntervall zurück wird für Zeit auslesen gebraucht

Rückgabe

int Stepintervall in ms

Autor

Rupert

5.11.3.10 void Game::handleCollisions () [private]

Kollisionen in der Liste collisionsToHandle werden der Reihe nach aus Sicht des affectedObjects bearbeitet. In einer Schleife wird das jeweils erst CollisionEvent bearbeitet. Dabei werden nur an dem Objekt affectedObject Änderungen vorgenommen. Mögliche Objekte: Spieler(player), Gegner(enemy), Bierkrug(shot) mögliche Kollision mit Spieler(player), Hindernis(obstacle), Gegner(enemy), Bierkrug(shot), Power-Up(powerUp)

Autor

Johann (15.6.15)

5.11.3.11 void Game::loadLevelFile (QString *fileSpecifier*) [private]

Level-Datei auslesen.

Parameter

<i>fileSpecifier</i>	Diese Funktion liest Level-Dateien aus. In der Leveldatei werden Keywords für die anzulegen- den Objekte verwendet. Nach den Objekten stehen durch Kommata getrennt die benötigten Parameter. Ein Player-Eintrag enthält posX und posY. Ein Enemy-Eintrag enthält posX, posY und speedX. Ein Obstacle-Eintrag enthält posX und posY. Ein Plane-Eintrag (Zwischenebene) enthält posX und posY. Ein PowerUp-Eintrag enthält posX, posY und die jeweiligen Boni.
----------------------	---

Autor

Simon

5.11.3.12 void Game::menuInit () [private]

Funktionen zu Start und Ende der Applikation.

Initialisierung der Menüs wird in [start\(\)](#) aufgerufen Logik: Startmenü Credits Levelauswahl spielen... Pause Name eingeben Spielstatistik Highscore Von vorne.

Autor

Rupert

5.11.3.13 void Game::reduceWorldObjects (Player * *playerPointer*) [private]

[Game::reduceWorldObjects](#).

Parameter

<i>playerPointer</i>	Alle Objekte aus der Liste <code>objectsToDelete</code> werden in der <code>wolrdObjects</code> gesucht und entfernt. Ihr Speicher wird wieder freigegeben. Die Funktion <code>reduceWorldObjects</code> löscht die <code>Game-Objects</code> und gibt den Speicher wieder frei, von denen der Spieler bereits weiter rechts als die <code>spawnDistance</code> entfernt ist.
----------------------	---

Autor

Simon, Johann

5.11.3.14 `void Game::renderGraphics (std::list< GameObject * > * objectList, Player * playerPointer)` [private]

[Game::renderGraphics](#) Positionssaktualisierungen der Grafiken aller Beewglichen Objekte.

Parameter

<i>objectList</i>	
-------------------	--

5.11.3.15 `void Game::setState (enum gameState newState)`

Hilfsfunktion.

setzt den Spielstatus

Parameter

<i>newState</i>	
-----------------	--

Autor

Rupert

5.11.3.16 `int Game::start ()`

Starten der Applikation.

Die Startfunktion, erstellt Fenster und Menüs, wird von `main()` aufgerufen Grafik und [Input](#) (Flo, Felix): Erstelle Q-Application app mit `QGraphicsView` Widget `window` (Eventfilter installiert) und Zeiger `input` auf [Input](#) Objekt. Um Funktionen der Tastatur Eingabe entwickeln zu können ist ein Qt Widget Fenster nötig. Auf dem Widget wird ein Eventfilter installiert welcher kontinuierlich Tastatureingaben mitloggt. Die Eingaben werden in dem Objekt der [Input](#) Klasse gespeichert und können über `getKeyactions()` abgerufen werden.

Logik (Rupert): Außerdem wird ein Timer gestartet, der in jedem Intervall `timerEvent(...)` aufruft, wo dann [step\(\)](#) aufgerufen wird. Das ist dann unsere Game-Loop. Der Timer funktioniert auch bei 5ms Intervall noch genau. Menüs

(Rupert): Alle Menüs werden angelegt

`gameState` wird auf `gameMenuStart` gesetzt, dh das Spiel startet im Startmenü

Rückgabe

Rückgabewert von `app.exec()`

Autor

Rupert, Felix

5.11.3.17 void Game::startNewGame (QString *levelFileName*, int *levelNum*) [private]

Level starten und beenden.

Startet neues Spiel lädt Leveldatei füllt worldobjects.

5.11.3.18 int Game::step ()

Game-Loop Diese Funktion wird von [timerEvent\(\)](#) aufgerufen und ist für den kompletten Ablauf des Spiels verantwortlich. grober Ablauf: LOOP:

- Timer starten
- Neue Objekte zur Welt hinzufügen
- alte Objekte löschen
- [Input](#) auslesen
- Bewegungen berechnen
- Kollisionskontrolle
- Bewegungen korrigieren
- Events behandeln (Treffer..)
- Grafik rendern und ausgeben
- [Audio](#) ausgeben
- verbleibende Zeit im Slot berechnen (Timer auslesen)
- entsprechend warten goto LOOP

Rückgabe

0 bei fehlerfreiem Beenden

Autor

Rupert, Felix

[Noch zu erledigen](#) Erfolgreich Schriftzug einfügen

[Noch zu erledigen](#) GameOver schriftzug einfügen

5.11.3.19 void Game::timerEvent (QTimerEvent * *event*) [protected]

wird regelmäßig aufgerufen event muss drinstehen, damit der Timer die Funktion aufruft

Parameter

<i>event</i>	
--------------	--

Autor

Rupert, Felix

5.11.3.20 void Game::updateAudioevents () [private]

durchläuft die Liste audioStorage und zählt die [Audio](#) Cooldowns herunter. Die Soundevents die noch laufen, werden an die Liste AudioEvents übergeben. Die fertigen werden gelöscht.

Autor

Johann, Felix

5.11.3.21 void Game::updateHighScore (std::string mode) [private]

[Game::updateHighScore](#) Diese Funktion liest und aktualisiert die Highscore des Spiels. Als Parameter wird ein std::string mode erwartet. Ist der mode = "write", so wird die aktuelle Highscore unter Berücksichtigung der aktuellen playerScore neu geschrieben. Alle anderen Werte für mode lesen nur die alte Highscore und die des Spielers in die Liste ein, um sie z.B. im Highscore-Menü anzuzeigen. Dazu wird versucht, die Datei "wiesnHighscore.txt" aus-zulesen. Ist dies nicht möglich, so wurde das Spiel in dem aktuellen Verzeichnis noch nie gestartet. Falls die Datei gefunden und gelesen werden kann, so wird jeder Highscore-Eintrag in die scoreList aufgenommen. Anschließend wird die Liste nach der Summe der Punkte absteigend sortiert, und nur die 10 besten Elemente werden gespeichert. Wurde für das aktuelle Spiel eine Score angelegt und in der scoreList gespeichert, so wird dieser Eintrag eingeordnet und gegebenenfalls auch abgespeichert.

Autor

Simon

5.11.3.22 void Game::updateScore () [private]

[Game::updateScore](#) Aktualisiert die Score des Spielers. Diese Score wird von der Grafik während des Spiels ausgegeben und am Ende des Spiels in die Highscore aufgenommen.

Autor

Simon

5.11.4 Dokumentation der Datenelemente

5.11.4.1 std::list<GameObject*> Game::levelSpawn [private]

Statische Objekte, die zu Anfang gespawnt werden.

Objekte die zur Laufzeit dynamisch gespawnt werden

5.11.4.2 int Game::stepCount = 0 [private]

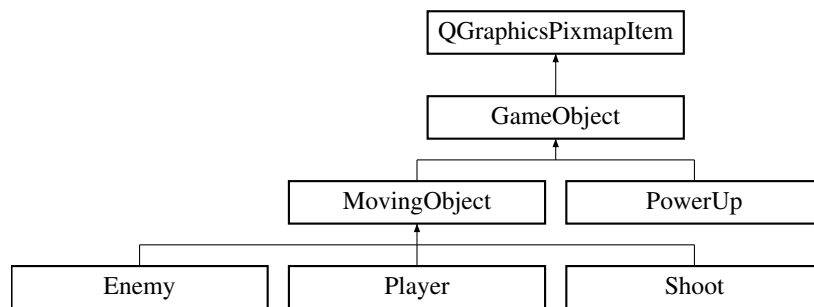
stepCount wird mit jedem Step um ein erhöht Auslesen der vergangenen Zeit: stepCount * [getStepIntervall\(\)](#)

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- Wiesn-Run/src/game.h
- Wiesn-Run/src/game.cpp

5.12 GameObject Klassenreferenz

Klassendiagramm für GameObject:



Öffentliche Methoden

- **GameObject** (int posX, int posY, int length, int height, objectType type)
GameObject::GameObject Konstruktor.
- **GameObject** (int posX, int posY, objectType type)
- int **getPosX** () const
- int **getPosY** () const
- int **getLength** () const
- int **getHeight** () const
- objectType **getType** () const
- void **setAudioID** (int audioID)
- int **getAudioID** () const

Geschützte Attribute

- int **posX**
- int **posY**

Private Attribute

- int **length**
- int **height**
- objectType **type**
- int **audioID**

5.12.1 Beschreibung der Konstruktoren und Destruktoren

5.12.1.1 GameObject::GameObject (int posX, int posY, int length, int height, objectType type)

GameObject::GameObject Konstruktor.

Parameter

<i>length</i>	: Länge
<i>height</i>	: Höhe
<i>type</i>	: Typ
<i>posX</i>	: X-Position
<i>posY</i>	: Y-Position

<code>colType</code>	: Kollisionstyp
----------------------	-----------------

Autor

Johann

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- Wiesn-Run/src/gameobject.h
- Wiesn-Run/src/gameobject.cpp

5.13 Input Klassenreferenz

Die Input-Klasse aktualisiert die für das Spiel relevanten Tastatureingaben. Eine Instanz dieser Klasse wird innerhalb der [game.h](#) angelegt.

```
#include <input.h>
```

Öffentliche Typen

- enum [Keyaction](#) {
noKeyaction, Right, Up, Down,
Jump_Right, Shoot, Exit, Enter }
Keyaction definiert alle auszuwertenden Tastenkombinationen Bezeichner.
- enum [Keyletter](#) {
noKeyletter, a = (int)'a', b = (int)'b', c = (int)'c',
d = (int)'d', e = (int)'e', f = (int)'f', g = (int)'g',
h = (int)'h', i = (int)'i', j = (int)'j', k = (int)'k',
l = (int)'l', m = (int)'m', n = (int)'n', o = (int)'o',
p = (int)'p', q = (int)'q', r = (int)'r', s = (int)'s',
t = (int)'t', u = (int)'u', v = (int)'v', w = (int)'w',
x = (int)'x', y = (int)'y', z = (int)'z', A = (int)'A',
B = (int)'B', C = (int)'C', D = (int)'D', E = (int)'E',
F = (int)'F', G = (int)'G', H = (int)'H', I = (int)'I',
J = (int)'J', K = (int)'K', L = (int)'L', M = (int)'M',
N = (int)'N', O = (int)'O', P = (int)'P', Q = (int)'Q',
R = (int)'R', S = (int)'S', T = (int)'T', U = (int)'U',
V = (int)'V', W = (int)'W', X = (int)'X', Y = (int)'Y',
Z = (int)'Z', Backspace = (int)'\b' }
Keyletter definiert alle auszuwertenden Tastatur Buchstaben.

Öffentliche Methoden

- [Input](#) ()
Konstruktor instanziert ein Objekt der Klasse [Input](#).
- [~Input](#) ()
Destruktor löscht ein Objekt der Klasse [Input](#).
- void [evaluateKeyEvent](#) (QEvent *event)
Nach Aufruf über [Game::eventFilter](#) wertet [evaluateKeyEvent](#) alle im Moment gleichzeitig gepressten Tastatur Eingaben aus und speichert die zugehörigen enum ids in der Instanzvariable keyevents. Wird eine Taste nicht mehr gedrückt wird die enum id in keyevents gelöscht. Wird eine Taste neu gedrückt wird die enum id in keyevents hinzugefügt.
- QSet< int > [getKeyactions](#) ()

getKeyactions gibt bei Aufruf das QSet *keyactions* zurück, welches alle im Moment gedrückten Spielaktionen als Enum beinhaltet. Jeder Tastaturkombination wird eine Integer ID zugeordnet welche im QSet *keyactions* gespeichert ist. Über die Enumeration *Input::Keyaction* ist jeder Spielbefehl mit dem zugehörigen Indize in *keyactions* verknüpft. Möchte man nun beispielsweise abfragen ob der Spieler im Moment schießt so überprüft man: `input->getKeyactions().contains(Input::Keyaction::Shoot) == True`.

- `std::set< char > getKeyletters ()`

getKeyletters gibt bei Aufruf das QSet *keyletters* zurück, welches alle im Moment gedrückten Buchstaben als Enum beinhaltet. Jeder Buchstaben Taste wird ein String Buchstaben zugeordnet, welcher im QSet *keyletters* gespeichert ist. Über die Enumeration *Input::Keyletter* ist jeder Buchstabe mit dem zugehörigen Indize in *keyletters* verknüpft. Möchte man nun beispielsweise abfragen ob der Spieler im Moment die "a" Taste drückt so überprüft man: `input->getKeyletters().find(Input::Keyletter::a) != getKeyletters().end()`. Möchte man abfragen ob der Spieler im Moment die "A" Taste drückt so überprüft man: `input->getKeyletters().find(Input::Keyletter::A) != getKeyletters().end()`. Ist die Taste gedrückt so kann aus dem Enum *Keyletter* über eine Typenumwandlung der Char berechnet werden: `'a' = (char)Keyletter::a`

- `Keyaction getLastKeyaction ()`

Gibt letzte gedrückte Spielaktion als Enum *Keyaction* zurück und setzt die Variable *lastKeyaction* auf *noKeyaction*. Wird für die Menüführung gebraucht, da ein dauerhaftes Auswerten der Tasten dort zu Sprüngen beim Auswählen der Menü Einträge führt.

- `Keyletter getLastKeyletter ()`

Gibt letzten gedrückten Buchstaben als enum *Keyletter* zurück und setzt die Variable *lastKeyletter* auf *noKeyletter*. Wurde eine Taste gedrückt (`lastKeyletter_return != noKeyletter`) so kann aus dem Enum *Keyletter* über eine Typenumwandlung der zugehörige Char berechnet werden: `a = (char)lastKeyletter_return`. Verwendung findet die Funktion bei der Eingabe des Highscore Namens.

Private Methoden

- `void updateKeys ()`

updateKeyactions berechnet aus allen in *keyevents* gespeicherten Tastatureingaben die für das Spiel relevanten Kombinationen und speichert diese in *keyactions*. Jede Aktionen ist im QSet *keyactions* als Integer gespeichert, welche über die enumeration *Keyaction* adressiert wird. Wird durch die Funktion *eventFilter* ein *KeyRelease* oder *KeyPress* Event aufgezeichnet, so wird der QSet *keyactions* gelöscht und mit den aktualisierten Werten im Qset *keyevents* abgeglichen. Sind Tasten oder Tastenkombinationen gedrückt worden, welche für das Spiel relevant sind so wird die zur Aktion gehörige integer ID im QSet *keyactions* hinzugefügt.

Private Attribute

- `QSet< int > keyevents`

keyevents speichert die id aller im Momment gepressten Tasten.

- `QSet< int > keyactions`

Die Variable *keyactions* speichert die id aller im Moment gepressten Tastenkombinationen, welche für das Spiel relevant sind.

- `std::set< char > keyletters`

Die Variable *keyletters* speichert die die Buchstababen als "strings" aller im Moment gepressten Buchstaben Tasten.

- `Keyaction lastKeyaction`

Die Variable *lastKeyaction* speichert die letzte gedrückte Tastenkombination als Enum *Keyaction*.

- `Keyletter lastKeyletter`

Die Variable *lastKeyletter* speichert den letzten gedrückten Buchstaben als Enum *Keyletter*.

5.13.1 Ausführliche Beschreibung

Die Input-Klasse aktualisiert die für das Spiel relevanten Tastatureingaben. Eine Instanz dieser Klasse wir innerhalb der *game.h* angelegt.

Autor

Felix Pfreundtner

5.13.2 Beschreibung der Konstruktoren und Destruktoren

5.13.2.1 Input::Input ()

Konstruktor instanziert ein Objekt der Klasse [Input](#).

Autor

Felix Pfreundtner

5.13.2.2 Input::~~Input ()

Destruktor löscht ein Objekt der Klasse [Input](#).

Autor

Felix Pfreundtner

5.13.3 Dokumentation der Elementfunktionen

5.13.3.1 void Input::evaluateKeyEvent (QEvent * event)

Nach Aufruf über [Game::eventFilter](#) wertet evaluateKeyEvent alle im Moment gleichzeitig gepressten Tastatur Eingaben aus und speichert die zugehörigen enum ids in der Instanzvariable keyevents. Wird eine Taste nicht mehr gedrückt wird die enum id in keyevents gelöscht. Wird eine Taste neu gedrückt wird die enum id in keyevents hinzugefügt.

Parameter

<i>QEvent</i>	*event
---------------	--------

Autor

Felix Pfreundtner

5.13.3.2 QSet< int > Input::getKeyactions ()

getKeyactions gibt bei Aufruf das QSet keyactions zurück, welches alle im Moment gedrückten Spielaktionen als Enum beinhaltet. Jeder Tastaturkombination wird eine Integer ID zugeordnet welche im QSet keyactions gespeichert ist. Über die Enumeration [Input::Keyaction](#) ist jeder Spielbefehl mit dem zugehörigen Indize in keyactions verknüpft. Möchte man nun beispielsweise abfragen ob der Spieler im Moment schießt so überprüft man: `input->getKeyactions().contains(Input::Keyaction::Shoot) == True`.

Rückgabe

QSet<int> Instanzvariable keyactions

Autor

Felix Pfreundtner

5.13.3.3 `std::set< char > Input::getKeyletters ()`

`getKeyletters` gibt bei Aufruf das `QSet` `keyletters` zurück, welches alle im Moment gedrückten Buchstaben als Enum beinhaltet. Jeder Buchstaben Taste wird ein String Buchstaben zugeordnet, welcher im `QSet` `keyletters` gespeichert ist. Über die Enumeration `Input::Keyletter` ist jeder Buchstabe mit dem zugehörigen Indize in `keyletters` verknüpft. Möchte man nun beispielsweise abfragen ob der Spieler im Moment die "a" Taste drückt so überprüft man: `input->getKeyletters().find(Input::Keyletter::a) != getKeyletters().end()`. Möchte man abfragen ob der Spieler im Moment die "A" Taste drückt so überprüft man: `input->getKeyletters().find(Input::Keyletter::A) != getKeyletters().end()`. Ist die Taste gedrückt so kann aus dem Enum `Keyletter` über eine Typenumwandlung der Char berechnet werden: `'a' = (char)Keyletter::a`

Rückgabe

`std::set<char>` Instanzvariable `keyletters`

Autor

Felix Pfreundtner

5.13.3.4 `Input::Keyaction Input::getLastKeyaction ()`

Gibt letzte gedrückte Spielaktion als Enum `Keyaction` zurück und setzt die Variable `lastKeyaction` auf `noKeyaction`. Wird für die Menüführung gebraucht, da ein dauerhaftes Auswerten der Tasten dort zu Sprüngen beim Auswählen der Menü Einträge führt.

Rückgabe

Enum `Keyaction` Instanzvariable `lastKeyaction`

Autor

Rupert, Felix

5.13.3.5 `Input::Keyletter Input::getLastKeyletter ()`

Gibt letzten gedrückten Buchstaben als enum `Keyletter` zurück und setzt die Variable `lastKeyletter` auf `noKeyletter`. Wurde eine Taste gedrückt (`lastKeyletter_return != noKeyletter`) so kann aus dem Enum `Keyletter` über eine Typenumwandlung der zugehörige Char berechnet werden: `a = (char)lastKeyletter_return`. Verwendung findet die Funktion bei der Eingabe des Highscore Namens.

Rückgabe

Enum `Keyletter` Instanzvariable `lastKeyletter`

Autor

Felix

5.13.3.6 `void Input::updateKeys () [private]`

`updateKeyactions` berechnet aus allen in `keyevents` gespeicherten Tastatureingaben die für das Spiel relevanten Kombinationen und speichert diese in `keyactions`. Jede Aktionen ist im `QSet` `keyactions` als Integer gespeichert, welche über die enumeration `Keyaction` adressiert wird. Wird durch die Funktion `eventFilter` ein `KeyRelease` oder `KeyPress` Event aufgezeichnet, so wird der `QSet` `keyactions` gelöscht und mit den aktualisierten Werten im `Qset` `keyevents` abgeglichen. Sind Tasten oder Tastenkombinationen gedrückt worden, welche für das Spiel relevant sind so wird die zur Aktion gehörige integer ID im `QSet` `keyactions` hinzugefügt.

Autor

Felix Pfreundtner

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- Wiesn-Run/src/input.h
- Wiesn-Run/src/input.cpp

5.14 Menu Klassenreferenz

Menü-Klasse eine Instanz repräsentiert ein Menü mit diesen Funktionen:

```
#include <menu.h>
```

Klassen

- struct `menuEntry`
Struct zur Beschreibung eines Menü-Eintrags.

Öffentliche Typen

- enum `menuSelectionChange` { **up**, **down** }
wird von der Menu-Klasse zur Auswahl-Änderung benötigt
- enum `menuType` { **normal**, **highscore** }
für verschiedene Menü-Typen (für Background-Musik)

Öffentliche Methoden

- `Menu` (std::string *menuTitle, `menuType` type=normal)
Menu-Konstruktor.
- void `clear` ()
löscht alle Einträge bis auf den ersten
- `menuType` `getType` ()
gibt den Typ zurück
- std::string * `getTitle` ()
gibt den Titel zurück
- int `displayInit` ()
Initialisiert das angezeigt Menü
- int `displayUpdate` ()
Aktualisiert das angezeigt Menü
- int `addEntry` (std::string name, int id, bool clickable=false, gameState stateOnClick=(gameState) NULL)
Neuen Eintrag hinzufügen (evtl private -> Einträge nur im Konstruktor erstellen -> unterschiedlich viele Argumente)
- int `changeSelection` (`menuSelectionChange` changeType)
wird nach Tastendruck aufgerufen
- `Menu::menuEntry` * `getSelection` ()
Zeiger auf aktuelle gewählten Menüeintrag, sollte nach Enter aufgerufen werden.
- `Menu::menuEntry` * `getEntry` (int position)
Gibt Menü-Eintrag an der entsprechenden Position zurück.

Öffentliche Attribute

- QGraphicsPixmapItem * **background**
Zeiger auf die Menü-Scene und das Menü-Hintergrundbild.
- QGraphicsScene * **menuScene**

Private Methoden

- int **selectFirstEntry** ()
aktiviert ersten klickbaren Eintrag

Private Attribute

- std::list< struct **menuEntry** * > **menuEntrys**
Liste, die die Menü-Einträge enthält.
- int **currentPosition** = 0
Zeiger auf gewählten Menüpunkt.
- int **numberOfEntrys** = 0
Anzahl der Einträge.
- std::string * **title**
Zeiger auf String, in dem der Titel des Menüs steht.
- **menuType** type
Menü-Typ.

5.14.1 Ausführliche Beschreibung

Menü-Klasse eine Instanz repräsentiert ein Menü mit diesen Funktionen:

- Einträge hinzufügen
- aktuelle Auswahl ändern (nach Tastendruck)
- anzeigen

5.14.2 Beschreibung der Konstruktoren und Destruktoren

5.14.2.1 Menu::Menu (std::string * *menuTitle*, menuType *type* = normal)

Menu-Konstruktor.

Parameter

<i>Zeiger</i>	auf String mit Menu-Titel
---------------	---------------------------

Autor

Rupert

5.14.3 Dokumentation der Elementfunktionen

5.14.3.1 int Menu::addEntry (std::string *name*, int *id*, bool *clickable* = false, gameState *stateOnClick* = (gameState) NULL)

Neuen Eintrag hinzufügen (evtl private -> Einträge nur im Konstruktor erstellen -> unterschiedlich viele Argumente)

Neuen Eintrag hinzufügen.

Parameter

<i>name</i>	String, der angezeigt wird
<i>id</i>	zur eindeutigen Identifizierung, kann zB aus enum gecastet werden

Rückgabe

0 bei Erfolg

Autor

Rupert

5.14.3.2 int Menu::changeSelection (menuSelectionChange *changeType*)

wird nach Tastendruck aufgerufen

Parameter

<i>changeType</i>	entweder up oder down
-------------------	-----------------------

Rückgabe

0 bei Erfolg, -1 wenn kein klickbarer Eintrag vorhanden

Autor

Rupert

5.14.3.3 void Menu::clear ()

löscht alle Einträge bis auf den ersten

entfernt alle Einträge aus dem Menü Titel wird danach wieder hinzugefügt wird für Statistik und Highscore benötigt

5.14.3.4 int Menu::displayInit ()

Initialisiert das angezeigte Menü

Initialisiert das sichtbare Menü , muss immer nach anlegen der Menü Entrys aufgerufen werden.

Rückgabe

0 bei Erfolg

Autor

Flo

5.14.3.5 int Menu::displayUpdate ()

Aktualisiert das angezeigte Menü

aktualisiert das sichtbare Menü

Rückgabe

0 bei Erfolg

Autor

Flo

5.14.3.6 struct Menu::menuEntry * Menu::getEntry (int *position*)

Gibt Menü-Eintrag an der entsprechenden Position zurück.

gibt Eintrag an der gesuchten Position zurück

Parameter

<i>position</i>	
-----------------	--

Rückgabe

Zeiger auf gefundenen Eintrag, sonst NULL

Autor

Rupert

Schleife startet beim ersten Element und geht bis zum letzten Element durch

5.14.3.7 struct Menu::menuEntry * Menu::getSelection ()

Zeiger auf aktuelle gewählten Menüeintrag, sollte nach Enter aufgerufen werden.

gibt den gewählten Eintrag zurück sollte nach Enter aufgerufen werden

Rückgabe

Zeiger auf [menuEntry](#) des aktuellen Eintrags, NULL bei Fehler

Autor

Rupert

5.14.3.8 std::string * Menu::getTitle ()

gibt den Titel zurück

gibt den Menü-Titel zurück

Rückgabe

Zeiger auf String

Autor

Rupert

5.14.3.9 Menu::menuType Menu::getType ()

gibt den Typ zurück

gibt den Menü-Typ zurück

Rückgabe

enum menuType

Autor

Rupert

5.14.3.10 int Menu::selectFirstEntry () [private]

aktiviert ersten klickbaren Eintrag

Rückgabe

int 0 bei Erfolg, -1 sonst

Autor

Rupert

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- Wiesn-Run/src/menu.h
- Wiesn-Run/src/menu.cpp

5.15 Menu::menuEntry Strukturreferenz

Struct zur Beschreibung eines Menü-Eintrags.

```
#include <menu.h>
```

Öffentliche Attribute

- std::string **name**
- int **id**
- int **position**
- bool **isClickable**
- bool **menuOnEnter**
- gameState **stateOnClick**
- QGraphicsTextItem **showEntry**

5.15.1 Ausführliche Beschreibung

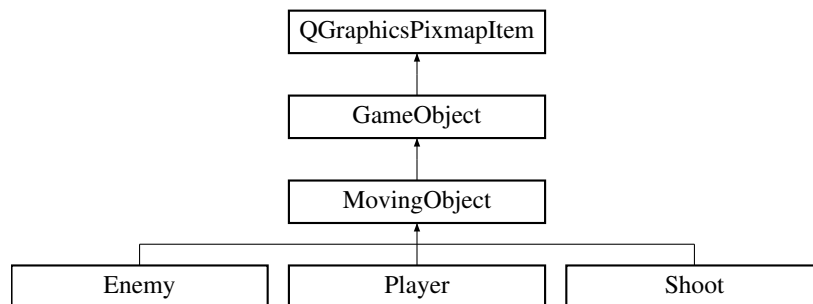
Struct zur Beschreibung eines Menü-Eintrags.

Die Dokumentation für diese Struktur wurde erzeugt aufgrund der Datei:

- Wiesn-Run/src/menu.h

5.16 MovingObject Klassenreferenz

Klassendiagramm für MovingObject:



Öffentliche Methoden

- **MovingObject** (int posX, int posY, objectType type, int speedX, int speedY)
- void **setPosX** (int posX)
- void **setPosY** (int posY)
- int **getSpeedX** () const
- int **getSpeedY** () const
- void **setSpeedX** (int speedX)
- void **setSpeedY** (int speedY)
- void **setFramesDirection** (int framesDirection)
- int **getFramesDirection** ()
- virtual void **update** ()=0
- void **flipHorizontal** ()
spiegelt Grafiken an der Y-Achse
- void **swapImage** ()

Geschützte Methoden

- void **updatePosition** ()
überschreibt die X und Y Position gemäß SpeedXY.

Private Attribute

- int **speedX**
- int **speedY**
- int **framesDirection** = 0
- bool **imageState** = true

Weitere Geerbte Elemente

5.16.1 Dokumentation der Elementfunktionen

5.16.1.1 void MovingObject::flipHorizontal ()

spiegelt Grafiken an der Y-Achse

Autor

Flo

5.16.1.2 void MovingObject::updatePosition () [protected]

überschreibt die X und Y Position gemäß SpeedXY.

Autor

Rupert

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- Wiesn-Run/src/movingobject.h
- Wiesn-Run/src/movingobject.cpp

5.17 PaDeviceInfo Strukturreferenz

```
#include <portaudio.h>
```

Öffentliche Attribute

- int **structVersion**
- const char * **name**
- [PaHostApiIndex](#) **hostApi**
- int **maxInputChannels**
- int **maxOutputChannels**
- [PaTime](#) **defaultLowInputLatency**
- [PaTime](#) **defaultLowOutputLatency**
- [PaTime](#) **defaultHighInputLatency**
- [PaTime](#) **defaultHighOutputLatency**
- double **defaultSampleRate**

5.17.1 Ausführliche Beschreibung

A structure providing information and capabilities of PortAudio devices. Devices may support input, output or both input and output.

5.17.2 Dokumentation der Datenelemente

5.17.2.1 PaTime PaDeviceInfo::defaultHighInputLatency

Default latency values for robust non-interactive applications (eg. playing sound files).

5.17.2.2 PaTime PaDeviceInfo::defaultLowInputLatency

Default latency values for interactive performance.

5.17.2.3 PaHostApiIndex PaDeviceInfo::hostApi

note this is a host API index, not a type id

Die Dokumentation für diese Struktur wurde erzeugt aufgrund der Datei:

- Wiesn-Run/src/[portaudio.h](#)

5.18 PaHostApiInfo Strukturreferenz

```
#include <portaudio.h>
```

Öffentliche Attribute

- int [structVersion](#)
- [PaHostApiType](#) type
- const char * [name](#)
- int [deviceCount](#)
- [PaDeviceIndex](#) [defaultInputDevice](#)
- [PaDeviceIndex](#) [defaultOutputDevice](#)

5.18.1 Ausführliche Beschreibung

A structure containing information about a particular host API.

5.18.2 Dokumentation der Datenelemente

5.18.2.1 [PaDeviceIndex](#) [PaHostApiInfo::defaultInputDevice](#)

The default input device for this host API. The value will be a device index ranging from 0 to ([Pa_GetDeviceCount\(\)](#)-1), or [paNoDevice](#) if no default input device is available.

5.18.2.2 [PaDeviceIndex](#) [PaHostApiInfo::defaultOutputDevice](#)

The default output device for this host API. The value will be a device index ranging from 0 to ([Pa_GetDeviceCount\(\)](#)-1), or [paNoDevice](#) if no default output device is available.

5.18.2.3 int [PaHostApiInfo::deviceCount](#)

The number of devices belonging to this host API. This field may be used in conjunction with [Pa_HostApiDeviceIndexToDeviceIndex\(\)](#) to enumerate all devices for this host API.

Siehe auch

[Pa_HostApiDeviceIndexToDeviceIndex](#)

5.18.2.4 const char* [PaHostApiInfo::name](#)

A textual description of the host API for display on user interfaces.

5.18.2.5 int [PaHostApiInfo::structVersion](#)

this is struct version 1

5.18.2.6 [PaHostApiType](#) [PaHostApiInfo::type](#)

The well known unique identifier of this host API

Siehe auch

[PaHostApiTypeId](#)

Die Dokumentation für diese Struktur wurde erzeugt aufgrund der Datei:

- [Wiesn-Run/src/portaudio.h](#)

5.19 PaHostErrorInfo Strukturreferenz

```
#include <portaudio.h>
```

Öffentliche Attribute

- [PaHostApiTypeId](#) `hostApiType`
- `long` `errorCode`
- `const char *` `errorText`

5.19.1 Ausführliche Beschreibung

Structure used to return information about a host error condition.

5.19.2 Dokumentation der Datenelemente

5.19.2.1 `long PaHostErrorInfo::errorCode`

the error code returned

5.19.2.2 `const char* PaHostErrorInfo::errorText`

a textual description of the error if available, otherwise a zero-length string

5.19.2.3 `PaHostApiTypeId PaHostErrorInfo::hostApiType`

the host API which returned the error code

Die Dokumentation für diese Struktur wurde erzeugt aufgrund der Datei:

- [Wiesn-Run/src/portaudio.h](#)

5.20 PaStreamCallbackTimeInfo Strukturreferenz

```
#include <portaudio.h>
```

Öffentliche Attribute

- [PaTime](#) `inputBufferAdcTime`
- [PaTime](#) `currentTime`
- [PaTime](#) `outputBufferDacTime`

5.20.1 Ausführliche Beschreibung

Timing information for the buffers passed to the stream callback.

Time values are expressed in seconds and are synchronised with the time base used by [Pa_GetStreamTime\(\)](#) for the associated stream.

Siehe auch

[PaStreamCallback](#), [Pa_GetStreamTime](#)

5.20.2 Dokumentation der Datenelemente

5.20.2.1 **PaTime** [PaStreamCallbackTimeInfo::currentTime](#)

The time when the stream callback was invoked

5.20.2.2 **PaTime** [PaStreamCallbackTimeInfo::inputBufferAdcTime](#)

The time when the first sample of the input buffer was captured at the ADC input

5.20.2.3 **PaTime** [PaStreamCallbackTimeInfo::outputBufferDacTime](#)

The time when the first sample of the output buffer will output the DAC

Die Dokumentation für diese Struktur wurde erzeugt aufgrund der Datei:

- [Wiesn-Run/src/portaudio.h](#)

5.21 PaStreamInfo Strukturreferenz

```
#include <portaudio.h>
```

Öffentliche Attribute

- int [structVersion](#)
- [PaTime](#) [inputLatency](#)
- [PaTime](#) [outputLatency](#)
- double [sampleRate](#)

5.21.1 Ausführliche Beschreibung

A structure containing unchanging information about an open stream.

Siehe auch

[Pa_GetStreamInfo](#)

5.21.2 Dokumentation der Datenelemente

5.21.2.1 PaTime PaStreamInfo::inputLatency

The input latency of the stream in seconds. This value provides the most accurate estimate of input latency available to the implementation. It may differ significantly from the suggestedLatency value passed to [Pa_OpenStream\(\)](#). The value of this field will be zero (0.) for output-only streams.

Siehe auch

[PaTime](#)

5.21.2.2 PaTime PaStreamInfo::outputLatency

The output latency of the stream in seconds. This value provides the most accurate estimate of output latency available to the implementation. It may differ significantly from the suggestedLatency value passed to [Pa_OpenStream\(\)](#). The value of this field will be zero (0.) for input-only streams.

Siehe auch

[PaTime](#)

5.21.2.3 double PaStreamInfo::sampleRate

The sample rate of the stream in Hertz (samples per second). In cases where the hardware sample rate is inaccurate and PortAudio is aware of it, the value of this field may be different from the sampleRate parameter passed to [Pa_OpenStream\(\)](#). If information about the actual hardware sample rate is not available, this field will have the same value as the sampleRate parameter passed to [Pa_OpenStream\(\)](#).

5.21.2.4 int PaStreamInfo::structVersion

this is struct version 1

Die Dokumentation für diese Struktur wurde erzeugt aufgrund der Datei:

- [Wiesn-Run/src/portaudio.h](#)

5.22 PaStreamParameters Strukturreferenz

```
#include <portaudio.h>
```

Öffentliche Attribute

- [PaDeviceIndex](#) device
- int [channelCount](#)
- [PaSampleFormat](#) sampleFormat
- [PaTime](#) suggestedLatency
- void * [hostApiSpecificStreamInfo](#)

5.22.1 Ausführliche Beschreibung

Parameters for one direction (input or output) of a stream.

5.22.2 Dokumentation der Datenelemente

5.22.2.1 `int PaStreamParameters::channelCount`

The number of channels of sound to be delivered to the stream callback or accessed by [Pa_ReadStream\(\)](#) or [Pa_WriteStream\(\)](#). It can range from 1 to the value of `maxInputChannels` in the [PaDeviceInfo](#) record for the device specified by the device parameter.

5.22.2.2 `PaDeviceIndex PaStreamParameters::device`

A valid device index in the range 0 to ([Pa_GetDeviceCount\(\)](#)-1) specifying the device to be used or the special constant `paUseHostApiSpecificDeviceSpecification` which indicates that the actual device(s) to use are specified in `hostApiSpecificStreamInfo`. This field must not be set to `paNoDevice`.

5.22.2.3 `void* PaStreamParameters::hostApiSpecificStreamInfo`

An optional pointer to a host api specific data structure containing additional information for device setup and/or stream processing. `hostApiSpecificStreamInfo` is never required for correct operation, if not used it should be set to `NULL`.

5.22.2.4 `PaSampleFormat PaStreamParameters::sampleFormat`

The sample format of the buffer provided to the stream callback, [a_ReadStream\(\)](#) or [Pa_WriteStream\(\)](#). It may be any of the formats described by the `PaSampleFormat` enumeration.

5.22.2.5 `PaTime PaStreamParameters::suggestedLatency`

The desired latency in seconds. Where practical, implementations should configure their latency based on these parameters, otherwise they may choose the closest viable latency instead. Unless the suggested latency is greater than the absolute upper limit for the device implementations should round the `suggestedLatency` up to the next practical value - ie to provide an equal or higher latency than `suggestedLatency` wherever possible. Actual latency values for an open stream may be retrieved using the `inputLatency` and `outputLatency` fields of the [PaStreamInfo](#) structure returned by [Pa_GetStreamInfo\(\)](#).

Siehe auch

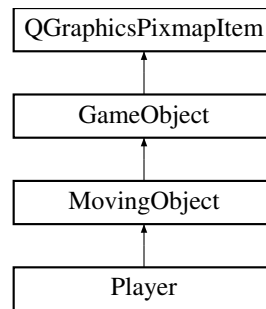
default*Latency in [PaDeviceInfo](#), *Latency in [PaStreamInfo](#)

Die Dokumentation für diese Struktur wurde erzeugt aufgrund der Datei:

- [Wiesn-Run/src/portaudio.h](#)

5.23 Player Klassenreferenz

Klassendiagramm für Player:



Öffentliche Methoden

- **Player** (int posX, int posY, int speedX)
- int **getHealth** () const
Player::getHealth Gibt aktuellen Lebensstand zurück.
- void **setHealth** (int health)
Player::setHealth Lebensstand des Spielers wird gesetzt.
- void **increaseHealth** (int health)
- bool **receiveDamage** (int damage)
Player::receiveDamage.
- int **getAlcoholLevel** () const
Player::getAlcoholLevel Gibt den Pegel des Spielers zurück.
- void **increaseAlcoholLevel** (int additionalAlcohol)
Player::increaseAlcoholLevel AlkoholPegel wird verändert. Durch einen negativen Wert im Argument wird der Pegel gesenkt.
- void **decreaseAlcoholLevel** (int decreaseLevel)
Player::decreaseAlcoholLevel verringert den Pegel des Spielers.
- int **getAmmunatiuon** () const
Player::getAmmunatiuon Gibt verbleibende Munition zurück.
- void **increaseAmmunation** (int ammunationBonus)
Player::increaseAmmunation erhöht die verbleibende Munition des Spielers um 1.
- void **decreaseAmmunation** ()
Player::decreaseAmmunation verringert die verbleibende Munition des Spielers um 1.
- void **setFireCooldown** ()
- int **getFireCooldown** ()
Player::getFireCooldown.
- int **getInflictedDamage** () const
Player::getInflictedDamage.
- int **getImmunityCooldown** () const
Player::getImmunityCooldown.
- void **setImmunityCooldown** (int remainingTime)
Wird nicht benutzt 23.6.
- void **startJump** ()
beginnt einen Sprung Nur wenn der Spieler sich nicht in der Luft befindet
- bool **inJump** () const
gibt den Sprung-Zustande des Spielers zurück
- void **resetJumpState** ()
Gibt an dass der Spieler nicht in einem Sprung ist.
- void **abortJump** ()
Methode wird aufgerufen, wenn der Spieler bei einem Sprung mit einem Hinderniss zusammengestoßen ist.
- int **getEnemiesKilled** ()

Player::getEnemiesKilled Übergibt die Zahl getöteter Gegner.

- void `increaseEnemiesKilled` ()

Perhöht die Anzahl der getöteten Gegner um 1.

- virtual void `update` ()

Player::update führt die Bewegung des Spielers aus (über `updatePosition`) und verringert *Cooldown-Variable*.

Private Attribute

- int **health**
- int **alcoholLevel**
- int **ammunation**
- int **inflictedDamage**
- int **immunityCooldown**
- int **fireCooldown**
- int **fireRate**
- bool **jumpState**
- int **jumpCooldown**
- int **enemiesKilled**

Weitere Geerbte Elemente

5.23.1 Beschreibung der Konstruktoren und Destruktoren

5.23.1.1 `Player::Player` (int *posX*, int *posY*, int *speedX*)

Class `Player` lastUpdate: `update()` 10.6 Johann

5.23.2 Dokumentation der Elementfunktionen

5.23.2.1 void `Player::decreaseAlcoholLevel` (int *decreaseLevel*)

`Player::decreaseAlcoholLevel` verringert den Pegel des Spielers.

Noch zu erledigen Überflüssig, da nie aufgerufen. Auch wenn der Name es nicht vermuten lässt: `increaseAlcoholLevel` kann den Level auch verringern und wird benutzt.

Parameter

<i>decreaseLevel</i>	Wert um den der Pegel verringert wird
----------------------	---------------------------------------

5.23.2.2 int `Player::getAlcoholLevel` () const

`Player::getAlcoholLevel` Gibt den Pegel des Spielers zurück.

Rückgabe

: Alkoholpegel

5.23.2.3 `int Player::getAmmunatiuon () const`

[Player::getAmmunatiuon](#) Gibt verbleibende Munition zurück.

Rückgabe

: verbleibende Munition

5.23.2.4 `int Player::getFireCooldown ()`

[Player::getFireCooldown](#).

Rückgabe

verbleibende Zeit bs nächster schuss möglich ist

5.23.2.5 `int Player::getHealth () const`

[Player::getHealth](#) Gibt aktuellen Lebensstand zurück.

Rückgabe

: Lebensstand

5.23.2.6 `int Player::getImmunityCooldown () const`

[Player::getImmunityCooldown](#).

Rückgabe

5.23.2.7 `int Player::getInflictedDamage () const`

[Player::getInflictedDamage](#).

Rückgabe

Schaden den der Spieler zufügt

5.23.2.8 `void Player::increaseAlcoholLevel (int additionalAlcohol)`

[Player::increaseAlcoholLevel](#) AlkoholPegel wird verändert. Durch einen negativen Wert im Argument wird der Pegel gesenkt.

Parameter

<i>additionalAlcohol</i>	Wert um den erhöht wird
--------------------------	-------------------------

5.23.2.9 `bool Player::inJump () const`

gibt den Sprung-Zustand des Spielers zurück

Rückgabe

5.23.2.10 `bool Player::receiveDamage (int damage)`

[Player::receiveDamage](#).

Rückgabe

Lebenszustand des Spielers: true = tot

5.23.2.11 `void Player::setHealth (int health)`

[Player::setHealth](#) Lebensstand des Spielers wird gesetzt.

Parameter

<i>health</i>	Lebensstand auf den der Spieler gesetzt wird
---------------	--

5.23.2.12 `void Player::setImmunityCooldown (int remainingTime)`

Wird nicht benutzt 23.6.

[Player::setImmunityCooldown](#) Zahl der Frames für Unverwundbarkeit wird gesetzt.

Parameter

<i>immunity-Cooldown</i>	Zahl der Frames
--------------------------	-----------------

5.23.2.13 `void Player::update () [virtual]`

[Player::update](#) führt die Bewegung des Spielers aus (über `updatePosition`) und verringert Cooldown-Variable.

Autor

Johann

Implementiert [MovingObject](#).

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- Wiesen-Run/src/player.h
- Wiesen-Run/src/player.cpp

5.24 `AudioControl::playStruct` Strukturreferenz

Öffentliche Attribute

- `int id`

- *id des `playStruct`*
- `audioType` *type*
 - *type des `playStruct`*
- `float volume`
 - *Lautstärke des `playStruct`.*
- `bool playnext`
 - *variable welche angibt ob sound im moment abgespielt wird*
- `Audio * audioobject`
 - *Zeiger auf das (Audio-)object des `playStruct`, welches Eventgruppe "type" zugeordnet ist.*
- `int position`
 - *aktuelle Abspielposition in Audiobjekt in Samples (Beginn des Abspielblockes mit Länge 1024 Samples*

Die Dokumentation für diese Struktur wurde erzeugt aufgrund der Datei:

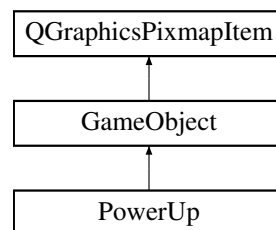
- Wiesn-Run/src/audiocontrol.h

5.25 PowerUp Klassenreferenz

Klasse für Power-Ups.

```
#include <powerup.h>
```

Klassendiagramm für PowerUp:



Öffentliche Methoden

- `PowerUp` (`int posX`, `int posY`, `int healthBonus`, `int alcoholLevelBonus`, `int ammuationBonus`, `int immunity-CooldownBonus`, `powerUpType type`)
 - *Konstruktor.*
- `~PowerUp` ()
 - *Destruktor.*
- `int getHealthBonus` () `const`
 - *Get-Methoden für die Objekteigenschaften.*
- `int getAlcoholLevelBonus` () `const`
 - *Gibt den Bonus auf Alcohollevel zurück.*
- `int getAmmuationBonus` () `const`
 - *Gibt den Bonus auf Munnition zurück.*
- `int getImmunityCooldownBonus` () `const`
 - *Gibt den Bonus auf Immunität zurück.*
- `powerUpType getPowerUPType` () `const`
 - *`PowerUp::getPowerUPType`.*

Private Attribute

- int **healthBonus**
- int **alcoholLevelBonus**
- int **ammunationBonus**
- int **immunityCooldownBonus**
- powerUpType **powType**

Weitere Geerbte Elemente

5.25.1 Ausführliche Beschreibung

Klasse für Power-Ups.

Autor

Johann

5.25.2 Beschreibung der Konstruktoren und Destruktoren

5.25.2.1 **PowerUp::PowerUp** (int *posX*, int *posY*, int *healthBonus*, int *alcoholLevelBonus*, int *ammunationBonus*, int *immunityCooldownBonus*, powerUpType *type*)

Konstruktor.

Parameter

<i>posX</i>	
<i>posY</i>	
<i>length</i>	
<i>height</i>	
<i>healthBonus</i>	
<i>alcoholLevel-</i> <i>Bonus</i>	
<i>ammunation-</i> <i>Bonus</i>	
<i>immunity-</i> <i>CooldownBonus</i>	

Autor

Johann

5.25.2.2 **PowerUp::~~PowerUp** ()

Destruktor.

Autor

Johann

5.25.3 Dokumentation der Elementfunktionen

5.25.3.1 int **PowerUp::getAlcoholLevelBonus** () const

Gibt den Bonus auf Alcohollevel zurück.

Autor

Johann

5.25.3.2 `int PowerUp::getAmmunationBonus () const`

Gibt den Bonus auf Munition zurück.

Autor

Johann

5.25.3.3 `int PowerUp::getHealthBonus () const`

Get-Methoden für die Objekteigenschaften.

Gibt den Bonus auf Leben zurück.

Autor

Johann

5.25.3.4 `int PowerUp::getImmunityCooldownBonus () const`

Gibt den Bonus auf Immunität zurück.

Autor

Johann

5.25.3.5 `powerUpType PowerUp::getPowerUPType () const`

[PowerUp::getPowerUPType](#).

Rückgabe

Art des powerups

Autor

Johann

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- Wiesn-Run/src/powerup.h
- Wiesn-Run/src/powerup.cpp

5.26 RenderBackground Klassenreferenz

Hintergrund-Klasse Eine Instanz wird bei jedem Levelstart in der Funktion [Game::startNewGame](#) angelegt. Die Klasse initialisiert alle Hintergrundgrafiken und aktualisiert deren Positionen im laufendem Spiel. Auch die Bewegungsparallaxe wird hier berechnet. Jede Hintergrundebene besteht immer aus zwei nebeneinander stehenden Bildern. Ist eines davon, bedingt durch die Vorwärtsbewegung des Spielers nicht mehr sichtbar, so wird es wieder am zweiten Bild vorbei, nach vorne geschoben. So wird gewährleistet das der Spieler nicht an den Bildern "vorbeiläuft".

```
#include <renderbackground.h>
```

Öffentliche Methoden

- [RenderBackground](#) (QGraphicsScene *scene, int level)
Konstruktor für alle Hintergrundgrafiken Hintergrundgrafiken werden initialisiert, positioniert und der Scene hinzugefügt.
- void [setPos](#) (int x, QGraphicsPixmapItem *background)
[RenderBackground::setPos](#) Funktion positioniert Hintergrundgrafiken neu.(nur "x" ändert sich, "y" ist immer 0)
- void [updateParallaxe](#) (int x)
[RenderBackground::updateParallaxe](#) Die Position der hinteren Hintergrundebene wird laufend so aktualisiert. Und zwar so dass sie sich mit halber Geschwindigkeit des Spielers bewegt und eine Parallaxeeffekt entsteht.
- void [updateBackgroundPos](#) (int x)
[RenderBackground::updateBackgroundPos](#) Immer wenn eine Hintergrundgrafik durch Spieler-Vorwärtsbewegung nicht mehr sichtbar ist wird sie wieder nach vorne, vor den Spieler versetzt. So ist ein ständig sichtbarer Hintergrund gewährleistet.

Private Attribute

- QGraphicsPixmapItem **backgroundOne**
- QGraphicsPixmapItem **backgroundTwo**
- QGraphicsPixmapItem **backgroundThree**
- QGraphicsPixmapItem **backgroundFour**
- int **imageLength** = 2560

5.26.1 Ausführliche Beschreibung

Hintergrund-Klasse Eine Instanz wird bei jedem Levelstart in der Funktion [Game::startNewGame](#) angelegt. Die Klasse initialisiert alle Hintergrundgrafiken und aktualisiert deren Positionen im laufendem Spiel. Auch die Bewegungsparallaxe wird hier berechnet. Jede Hintergrundebene besteht immer aus zwei nebeneinander stehenden Bildern. Ist eines davon, bedingt durch die Vorwärtsbewegung des Spielers nicht mehr sichtbar, so wird es wieder am zweiten Bild vorbei, nach vorne geschoben. So wird gewährleistet das der Spieler nicht an den Bildern "vorbeiläuft".

Autor

Flo

5.26.2 Beschreibung der Konstruktoren und Destruktoren

5.26.2.1 [RenderBackground::RenderBackground](#) (QGraphicsScene * scene, int level)

Konstruktor für alle Hintergrundgrafiken Hintergrundgrafiken werden initialisiert, positioniert und der Scene hinzugefügt.

Parameter

<i>scene</i>	: levelScene
<i>level</i>	: aktuelles Level

Autor

Flo

5.26.3 Dokumentation der Elementfunktionen

5.26.3.1 void [RenderBackground::setPos](#) (int x, QGraphicsPixmapItem * background)

[RenderBackground::setPos](#) Funktion positioniert Hintergrundgrafiken neu.(nur "x" ändert sich, "y" ist immer 0)

Parameter

<i>x</i>	: x-Position
<i>background</i>	: Hintergrundgrafikitem

Autor

Flo

5.26.3.2 void RenderBackground::updateBackgroundPos (int x)

[RenderBackground::updateBackgroundPos](#) Immer wenn eine Hintergrundgrafik durch Spieler-Vorwärtsbewegung nicht mehr sichtbar ist wird sie wieder nach vorne, vor den Spieler versetzt. So ist ein ständig sichtbarer Hintergrund gewährleistet.

Parameter

<i>x</i>	: x-Position des linken Bildrandes im Level
----------	---

Autor

Flo

5.26.3.3 void RenderBackground::updateParallaxe (int x)

[RenderBackground::updateParallaxe](#) Die Position der hinteren Hintergrundebene wird laufend so aktualisiert. Und zwar so dass sie sich mit halber Geschwindigkeit des Spielers bewegt und eine Parallaxeeffekt entsteht.

Parameter

<i>x</i>	: x-Wert der Positionsänderung des Spielers im aktuellen Step
----------	---

Autor

Flo

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- Wiesn-Run/src/renderbackground.h
- Wiesn-Run/src/renderbackground.cpp

5.27 RenderGUI Klassenreferenz

Anzeigen der Spielerwerte-Klasse Eine Instanz wird bei jedem Levelstart in der Funktion [Game::startNewGame](#) angelegt. Die Klasse initialisiert alle Grafikelemente die mit der Anzeige von Spielerwerten zu tun hat (Gesundheit, Alkoholpegel, Munitionsvorrat, Punkte). Außerdem werden hier auch die angezeigten Werte im Spiel fortlaufend aktualisiert. Alle Elemente sind "Kinder" der Gesundheitsanzeige um Positionsaktualisierungen zu vereinfachen (Kindelemente verhalten sich immer relativ um Elternobjekt und werden auch automatisch mit diesem der Scene hinzugefügt bzw. auch wieder entfernt)

```
#include <renderGUI.h>
```

Öffentliche Methoden

- [RenderGUI](#) (QGraphicsScene *scene)

Konstruktor für alle Spielerwert Anzeigen Die Grafikelemente der Anzeigen werden initialisiert, eingestellt und der Scene hinzugefügt.

- void **setPos** (int x)
RenderGUI::setPos sorgt für eine Positionsänderung identisch mit der des Spielers auf der X-Achse (Anzeigen bleiben auf den Spieler zentriert)
- void **setValues** (int health, int alcohol, int ammo, int score)
RenderGUI::setValues Aktualisierung aller angezeigten Wert, Gesundheits- und Pegelbalken sind immer auf die maximal möglichen Werte normiert.

Private Attribute

- QGraphicsTextItem **showHealth**
- QGraphicsRectItem **showHealthBar** [2]
- QGraphicsTextItem **showScore**
- QGraphicsTextItem **showAmmo**
- QGraphicsTextItem **showAlcohol**
- QGraphicsRectItem **showAlcoholBar** [2]

5.27.1 Ausführliche Beschreibung

Anzeigen der Spielerwerte-Klasse Eine Instanz wird bei jedem Levelstart in der Funktion [Game::startNewGame](#) angelegt. Die Klasse initialisiert alle Grafikelemente die mit der Anzeige von Spielerwerten zu tun hat (Gesundheit, Alkoholpegel, Munitionsvorrat, Punkte). Außerdem werden hier auch die angezeigten Werte im Spiel fortlaufend aktualisiert. Alle Elemente sind "Kinder" der Gesundheitsanzeige um Positionsaktualisierungen zu vereinfachen (Kindelemente verhalten sich immer relativ um Elternobjekt und werden auch automatisch mit diesem der Scene hinzugefügt bzw. auch wieder entfernt)

Autor

Flo

5.27.2 Beschreibung der Konstruktoren und Destruktoren

5.27.2.1 RenderGUI::RenderGUI (QGraphicsScene * scene)

Konstruktor für alle Spielerwert Anzeigen Die Grafikelemente der Anzeigen werden initialisiert, eingestellt und der Scene hinzugefügt.

Parameter

<i>scene</i>	: levelScene
--------------	--------------

Autor

Flo

5.27.3 Dokumentation der Elementfunktionen

5.27.3.1 void RenderGUI::setPos (int x)

[RenderGUI::setPos](#) sorgt für eine Positionsänderung identisch mit der des Spielers auf der X-Achse (Anzeigen bleiben auf den Spieler zentriert)

Parameter

<i>x</i>	: x-Wert der Positionsänderung des Spielers im aktuellen Step
----------	---

Autor

Flo

5.27.3.2 void RenderGUI::setValues (int *health*, int *alcohol*, int *ammo*, int *score*)

[RenderGUI::setValues](#) Aktualisierung aller angezeigten Wert, Gesundheits- und Pegelbalken sind immer auf die maximal möglichen Werte normiert.

Parameter

<i>health</i>	: aktueller Gesundheitswert
<i>alcohol</i>	: aktueller Alkoholpegelwert
<i>ammo</i>	: aktueller Munitionsstand
<i>score</i>	: aktueller Punktestad

Autor

Flo

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- Wiesn-Run/src/renderGUI.h
- Wiesn-Run/src/renderGUI.cpp

5.28 scoreStruct Strukturreferenz

Struktur für die Score des Spielers In dieser Struktur werden Name des Spielers, getötete Gegner, zurückgelegte Entfernung und Alkohol-Punkte gespeichert. Alkohol-Punkte erhält der Spieler für einen gewissen Pegel in einem Zeitabschnitt.

```
#include <definitions.h>
```

Öffentliche Attribute

- std::string **name**
- int **totalPoints**
- int **distanceCovered**
- int **alcoholPoints**
- int **enemiesKilled**

5.28.1 Ausführliche Beschreibung

Struktur für die Score des Spielers In dieser Struktur werden Name des Spielers, getötete Gegner, zurückgelegte Entfernung und Alkohol-Punkte gespeichert. Alkohol-Punkte erhält der Spieler für einen gewissen Pegel in einem Zeitabschnitt.

[Noch zu erledigen](#) Das Konzept der Alkohol-Punkte muss noch ausgearbeitet werden.

Autor

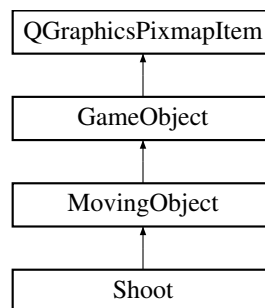
Simon

Die Dokumentation für diese Struktur wurde erzeugt aufgrund der Datei:

- Wiesn-Run/src/definitions.h

5.29 Shoot Klassenreferenz

Klassendiagramm für Shoot:



Öffentliche Methoden

- [Shoot](#) (int posX, int posY, int direction, objectType origin)
Konstruktor für einen Schuss(Bierkrug)
- int [getInflictedDamage](#) () const
[Shoot::getInflictedDamage](#) gibt den Schaden den der Schuss zufügt zurück.
- objectType [getOrigin](#) ()
[Shoot::getOrigin](#) gibt den Ursprung des Bierkrugs zurück, Wer hat ihn geworfen (Player/Enemy)
- virtual void **update** ()

Private Attribute

- int **inflictedDamage**
- objectType **origin**

Weitere Geerbte Elemente

5.29.1 Beschreibung der Konstruktoren und Destruktoren

5.29.1.1 Shoot::Shoot (int posX, int posY, int direction, objectType origin)

Konstruktor für einen Schuss(Bierkrug)

Parameter

<i>posX</i>	: x-Position
-------------	--------------

<i>posY</i>	: y-Position
<i>origin</i>	: Schuss Erzeuger

Schuss bewegt sich dreimal so schnell wie der spieler Größe des Bierkruges festgesetzt (erste idee)

Autor

Johann

5.29.2 Dokumentation der Elementfunktionen

5.29.2.1 int Shoot::getInflictedDamage () const

[Shoot::getInflictedDamage](#) gibt den Schaden den der Schuss zufügt zurück.

Rückgabe

Schaden

Autor

Johann

5.29.2.2 objectType Shoot::getOrigin ()

[Shoot::getOrigin](#) gibt den Ursprung des Bierkrugs zurück, Wer hat ihn geworfen (Player/Enemy)

Rückgabe

Ursprung des Bierkruges

Autor

Johann

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- Wiesen-Run/src/shoot.h
- Wiesen-Run/src/shoot.cpp

5.30 stateStruct Strukturreferenz

Struktur für die States des Spiels Sowohl Sound- als auch Grafik-Ausgabe erhalten aus den States Informationen darüber, was gerade im Spiel passiert, z.B. dass gerade der Spieler angreift, ein Gegner stirbt etc.

```
#include <definitions.h>
```

Öffentliche Attribute

- bool **gameOver** = false
- int **actLevel** = 0
- int **audioID_Background** = 0
- bool **beerCollected** = 0
- bool **chickenCollected** = 0

5.30.1 Ausführliche Beschreibung

Struktur für die States des Spiels Sowohl Sound- als auch Grafik-Ausgabe erhalten aus den States Informationen darüber, was gerade im Spiel passiert, z.B. dass gerade der Spieler angreift, ein Gegner stirbt etc.

Noch zu erledigen Diese Struktur ist vermutlich überflüssig.

Autor

Simon

Die Dokumentation für diese Struktur wurde erzeugt aufgrund der Datei:

- Wiesn-Run/src/definitions.h

Kapitel 6

Datei-Dokumentation

6.1 Wiesn-Run/src/portaudio.h-Dateireferenz

The portable PortAudio API.

Klassen

- struct [PaHostApiInfo](#)
- struct [PaHostErrorInfo](#)
- struct [PaDeviceInfo](#)
- struct [PaStreamParameters](#)
- struct [PaStreamCallbackTimeInfo](#)
- struct [PaStreamInfo](#)

Makrodefinitionen

- #define [paNoDevice](#) (([PaDeviceIndex](#))-1)
- #define [paUseHostApiSpecificDeviceSpecification](#) (([PaDeviceIndex](#))-2)
- #define [paFloat32](#) (([PaSampleFormat](#)) 0x00000001)
- #define [paInt32](#) (([PaSampleFormat](#)) 0x00000002)
- #define [paInt24](#) (([PaSampleFormat](#)) 0x00000004)
- #define [paInt16](#) (([PaSampleFormat](#)) 0x00000008)
- #define [paInt8](#) (([PaSampleFormat](#)) 0x00000010)
- #define [paUInt8](#) (([PaSampleFormat](#)) 0x00000020)
- #define [paCustomFormat](#) (([PaSampleFormat](#)) 0x00010000)
- #define [paNonInterleaved](#) (([PaSampleFormat](#)) 0x80000000)
- #define [paFormatIsSupported](#) (0)
- #define [paFramesPerBufferUnspecified](#) (0)
- #define [paNoFlag](#) (([PaStreamFlags](#)) 0)
- #define [paClipOff](#) (([PaStreamFlags](#)) 0x00000001)
- #define [paDitherOff](#) (([PaStreamFlags](#)) 0x00000002)
- #define [paNeverDropInput](#) (([PaStreamFlags](#)) 0x00000004)
- #define [paPrimeOutputBuffersUsingStreamCallback](#) (([PaStreamFlags](#)) 0x00000008)
- #define [paPlatformSpecificFlags](#) (([PaStreamFlags](#)) 0xFFFF0000)
- #define [paInputUnderflow](#) (([PaStreamCallbackFlags](#)) 0x00000001)
- #define [paInputOverflow](#) (([PaStreamCallbackFlags](#)) 0x00000002)
- #define [paOutputUnderflow](#) (([PaStreamCallbackFlags](#)) 0x00000004)
- #define [paOutputOverflow](#) (([PaStreamCallbackFlags](#)) 0x00000008)
- #define [paPrimingOutput](#) (([PaStreamCallbackFlags](#)) 0x00000010)

Typdefinitionen

- typedef int [PaError](#)
- typedef enum PaErrorCode **PaErrorCode**
- typedef int [PaDeviceIndex](#)
- typedef int [PaHostApiIndex](#)
- typedef enum [PaHostApiTypeId](#) [PaHostApiTypeId](#)
- typedef struct [PaHostApiInfo](#) [PaHostApiInfo](#)
- typedef struct [PaHostErrorInfo](#) [PaHostErrorInfo](#)
- typedef double [PaTime](#)
- typedef unsigned long [PaSampleFormat](#)
- typedef struct [PaDeviceInfo](#) [PaDeviceInfo](#)
- typedef struct [PaStreamParameters](#) [PaStreamParameters](#)
- typedef void [PaStream](#)
- typedef unsigned long [PaStreamFlags](#)
- typedef struct [PaStreamCallbackTimeInfo](#) [PaStreamCallbackTimeInfo](#)
- typedef unsigned long [PaStreamCallbackFlags](#)
- typedef enum [PaStreamCallbackResult](#) [PaStreamCallbackResult](#)
- typedef int [PaStreamCallback](#) (const void *input, void *output, unsigned long frameCount, const [PaStreamCallbackTimeInfo](#) *timeInfo, [PaStreamCallbackFlags](#) statusFlags, void *userData)
- typedef void [PaStreamFinishedCallback](#) (void *userData)
- typedef struct [PaStreamInfo](#) [PaStreamInfo](#)

Aufzählungen

- enum **PaErrorCode** {
paNoError = 0, **paNotInitialized** = -10000, **paUnanticipatedHostError**, **paInvalidChannelCount**,
paInvalidSampleRate, **paInvalidDevice**, **paInvalidFlag**, **paSampleFormatNotSupported**,
paBadIODeviceCombination, **paInsufficientMemory**, **paBufferTooBig**, **paBufferTooSmall**,
paNullCallback, **paBadStreamPtr**, **paTimedOut**, **paInternalError**,
paDeviceUnavailable, **paIncompatibleHostApiSpecificStreamInfo**, **paStreamIsStopped**, **paStreamIsNotStopped**,
paInputOverflowed, **paOutputUnderflowed**, **paHostApiNotFound**, **paInvalidHostApi**,
paCanNotReadFromACallbackStream, **paCanNotWriteToACallbackStream**, **paCanNotReadFromAnOutputOnlyStream**,
paCanNotWriteToAnInputOnlyStream, **paIncompatibleStreamHostApi**, **paBadBufferPtr** }
- enum [PaHostApiTypeId](#) {
paInDevelopment =0, **paDirectSound** =1, **paMME** =2, **paASIO** =3,
paSoundManager =4, **paCoreAudio** =5, **paOSS** =7, **paALSA** =8,
paAL =9, **paBeOS** =10, **paWDMKS** =11, **paJACK** =12,
paWASAPI =13, **paAudioScienceHPI** =14 }
- enum [PaStreamCallbackResult](#) { [paContinue](#) =0, [paComplete](#) =1, [paAbort](#) =2 }

Funktionen

- int [Pa_GetVersion](#) (void)
- const char * [Pa_GetVersionText](#) (void)
- const char * [Pa_GetErrorText](#) ([PaError](#) errorCode)
- [PaError](#) [Pa_Initialize](#) (void)
- [PaError](#) [Pa_Terminate](#) (void)
- [PaHostApiIndex](#) [Pa_GetHostApiCount](#) (void)
- [PaHostApiIndex](#) [Pa_GetDefaultHostApi](#) (void)
- const [PaHostApiInfo](#) * [Pa_GetHostApiInfo](#) ([PaHostApiIndex](#) hostApi)
- [PaHostApiIndex](#) [Pa_HostApiTypeIdToHostApiIndex](#) ([PaHostApiTypeId](#) type)

- [PaDeviceIndex](#) [Pa_HostApiDeviceIndexToDeviceIndex](#) ([PaHostApiIndex](#) hostApi, int hostApiDeviceIndex)
- const [PaHostErrorInfo](#) * [Pa_GetLastHostErrorInfo](#) (void)
- [PaDeviceIndex](#) [Pa_GetDeviceCount](#) (void)
- [PaDeviceIndex](#) [Pa_GetDefaultInputDevice](#) (void)
- [PaDeviceIndex](#) [Pa_GetDefaultOutputDevice](#) (void)
- const [PaDeviceInfo](#) * [Pa_GetDeviceInfo](#) ([PaDeviceIndex](#) device)
- [PaError](#) [Pa_IsFormatSupported](#) (const [PaStreamParameters](#) *inputParameters, const [PaStreamParameters](#) *outputParameters, double sampleRate)
- [PaError](#) [Pa_OpenStream](#) ([PaStream](#) **stream, const [PaStreamParameters](#) *inputParameters, const [PaStreamParameters](#) *outputParameters, double sampleRate, unsigned long framesPerBuffer, [PaStreamFlags](#) streamFlags, [PaStreamCallback](#) *streamCallback, void *userData)
- [PaError](#) [Pa_OpenDefaultStream](#) ([PaStream](#) **stream, int numInputChannels, int numOutputChannels, [PaSampleFormat](#) sampleFormat, double sampleRate, unsigned long framesPerBuffer, [PaStreamCallback](#) *streamCallback, void *userData)
- [PaError](#) [Pa_CloseStream](#) ([PaStream](#) *stream)
- [PaError](#) [Pa_SetStreamFinishedCallback](#) ([PaStream](#) *stream, [PaStreamFinishedCallback](#) *streamFinishedCallback)
- [PaError](#) [Pa_StartStream](#) ([PaStream](#) *stream)
- [PaError](#) [Pa_StopStream](#) ([PaStream](#) *stream)
- [PaError](#) [Pa_AbortStream](#) ([PaStream](#) *stream)
- [PaError](#) [Pa_IsStreamStopped](#) ([PaStream](#) *stream)
- [PaError](#) [Pa_IsStreamActive](#) ([PaStream](#) *stream)
- const [PaStreamInfo](#) * [Pa_GetStreamInfo](#) ([PaStream](#) *stream)
- [PaTime](#) [Pa_GetStreamTime](#) ([PaStream](#) *stream)
- double [Pa_GetStreamCpuLoad](#) ([PaStream](#) *stream)
- [PaError](#) [Pa_ReadStream](#) ([PaStream](#) *stream, void *buffer, unsigned long frames)
- [PaError](#) [Pa_WriteStream](#) ([PaStream](#) *stream, const void *buffer, unsigned long frames)
- signed long [Pa_GetStreamReadAvailable](#) ([PaStream](#) *stream)
- signed long [Pa_GetStreamWriteAvailable](#) ([PaStream](#) *stream)
- [PaHostApiTypeId](#) [Pa_GetStreamHostApiType](#) ([PaStream](#) *stream)
- [PaError](#) [Pa_GetSampleSize](#) ([PaSampleFormat](#) format)
- void [Pa_Sleep](#) (long msec)

6.1.1 Ausführliche Beschreibung

The portable PortAudio API.

6.1.2 Makro-Dokumentation

6.1.2.1 #define paClipOff ((PaStreamFlags) 0x00000001)

Disable default clipping of out of range samples.

Siehe auch

[PaStreamFlags](#)

6.1.2.2 #define paCustomFormat ((PaSampleFormat) 0x00010000)

Siehe auch

[PaSampleFormat](#)

6.1.2.3 `#define paDitherOff ((PaStreamFlags) 0x00000002)`

Disable default dithering.

Siehe auch

[PaStreamFlags](#)

6.1.2.4 `#define paFloat32 ((PaSampleFormat) 0x00000001)`

Siehe auch

[PaSampleFormat](#)

6.1.2.5 `#define paFormatIsSupported (0)`

Return code for `Pa_IsFormatSupported` indicating success.

6.1.2.6 `#define paFramesPerBufferUnspecified (0)`

Can be passed as the `framesPerBuffer` parameter to [Pa_OpenStream\(\)](#) or [Pa_OpenDefaultStream\(\)](#) to indicate that the stream callback will accept buffers of any size.

6.1.2.7 `#define paInputOverflow ((PaStreamCallbackFlags) 0x00000002)`

In a stream opened with `paFramesPerBufferUnspecified`, indicates that data prior to the first sample of the input buffer was discarded due to an overflow, possibly because the stream callback is using too much CPU time. Otherwise indicates that data prior to one or more samples in the input buffer was discarded.

Siehe auch

[PaStreamCallbackFlags](#)

6.1.2.8 `#define paInputUnderflow ((PaStreamCallbackFlags) 0x00000001)`

In a stream opened with `paFramesPerBufferUnspecified`, indicates that input data is all silence (zeros) because no real data is available. In a stream opened without `paFramesPerBufferUnspecified`, it indicates that one or more zero samples have been inserted into the input buffer to compensate for an input underflow.

Siehe auch

[PaStreamCallbackFlags](#)

6.1.2.9 `#define paInt16 ((PaSampleFormat) 0x00000008)`

Siehe auch

[PaSampleFormat](#)

6.1.2.10 #define paInt24 ((PaSampleFormat) 0x00000004)

Packed 24 bit format.

Siehe auch

[PaSampleFormat](#)

6.1.2.11 #define paInt32 ((PaSampleFormat) 0x00000002)

Siehe auch

[PaSampleFormat](#)

6.1.2.12 #define paInt8 ((PaSampleFormat) 0x00000010)

Siehe auch

[PaSampleFormat](#)

6.1.2.13 #define paNeverDropInput ((PaStreamFlags) 0x00000004)

Flag requests that where possible a full duplex stream will not discard overflowed input samples without calling the stream callback. This flag is only valid for full duplex callback streams and only when used in combination with the `paFramesPerBufferUnspecified` (0) `framesPerBuffer` parameter. Using this flag incorrectly results in a `paInvalidFlag` error being returned from `Pa_OpenStream` and `Pa_OpenDefaultStream`.

Siehe auch

[PaStreamFlags](#), [paFramesPerBufferUnspecified](#)

6.1.2.14 #define paNoDevice ((PaDeviceIndex)-1)

A special `PaDeviceIndex` value indicating that no device is available, or should be used.

Siehe auch

[PaDeviceIndex](#)

6.1.2.15 #define paNoFlag ((PaStreamFlags) 0)

Siehe auch

[PaStreamFlags](#)

6.1.2.16 #define paNonInterleaved ((PaSampleFormat) 0x80000000)

Siehe auch

[PaSampleFormat](#)

6.1.2.17 #define paOutputOverflow ((PaStreamCallbackFlags) 0x00000008)

Indicates that output data will be discarded because no room is available.

Siehe auch

[PaStreamCallbackFlags](#)

6.1.2.18 #define paOutputUnderflow ((PaStreamCallbackFlags) 0x00000004)

Indicates that output data (or a gap) was inserted, possibly because the stream callback is using too much CPU time.

Siehe auch

[PaStreamCallbackFlags](#)

6.1.2.19 #define paPlatformSpecificFlags ((PaStreamFlags)0xFFFF0000)

A mask specifying the platform specific bits.

Siehe auch

[PaStreamFlags](#)

6.1.2.20 #define paPrimeOutputBuffersUsingStreamCallback ((PaStreamFlags) 0x00000008)

Call the stream callback to fill initial output buffers, rather than the default behavior of priming the buffers with zeros (silence). This flag has no effect for input-only and blocking read/write streams.

Siehe auch

[PaStreamFlags](#)

6.1.2.21 #define paPrimingOutput ((PaStreamCallbackFlags) 0x00000010)

Some of all of the output data will be used to prime the stream, input data may be zero.

Siehe auch

[PaStreamCallbackFlags](#)

6.1.2.22 #define paUInt8 ((PaSampleFormat) 0x00000020)

Siehe auch

[PaSampleFormat](#)

6.1.2.23 #define paUseHostApiSpecificDeviceSpecification ((PaDeviceIndex)-2)

A special PaDeviceIndex value indicating that the device(s) to be used are specified in the host api specific stream info structure.

Siehe auch

[PaDeviceIndex](#)

6.1.3 Dokumentation der benutzerdefinierten Typen

6.1.3.1 typedef int PaDeviceIndex

The type used to refer to audio devices. Values of this type usually range from 0 to ([Pa_GetDeviceCount\(\)](#)-1), and may also take on the PaNoDevice and paUseHostApiSpecificDeviceSpecification values.

Siehe auch

[Pa_GetDeviceCount](#), [paNoDevice](#), [paUseHostApiSpecificDeviceSpecification](#)

6.1.3.2 typedef struct PaDeviceInfo PaDeviceInfo

A structure providing information and capabilities of PortAudio devices. Devices may support input, output or both input and output.

6.1.3.3 typedef int PaError

Error codes returned by PortAudio functions. Note that with the exception of paNoError, all PaErrorCodes are negative.

6.1.3.4 typedef int PaHostApiIndex

The type used to enumerate to host APIs at runtime. Values of this type range from 0 to ([Pa_GetHostApiCount\(\)](#)-1).

Siehe auch

[Pa_GetHostApiCount](#)

6.1.3.5 typedef struct PaHostApiInfo PaHostApiInfo

A structure containing information about a particular host API.

6.1.3.6 typedef enum PaHostApiTypeId PaHostApiTypeId

Unchanging unique identifiers for each supported host API. This type is used in the [PaHostApiInfo](#) structure. The values are guaranteed to be unique and to never change, thus allowing code to be written that conditionally uses host API specific extensions.

New type ids will be allocated when support for a host API reaches "public alpha" status, prior to that developers should use the paInDevelopment type id.

Siehe auch

[PaHostApiInfo](#)

6.1.3.7 typedef struct PaHostErrorInfo PaHostErrorInfo

Structure used to return information about a host error condition.

6.1.3.8 typedef unsigned long PaSampleFormat

A type used to specify one or more sample formats. Each value indicates a possible format for sound data passed to and from the stream callback, `Pa_ReadStream` and `Pa_WriteStream`.

The standard formats `paFloat32`, `paInt16`, `paInt32`, `paInt24`, `paInt8` and `aUInt8` are usually implemented by all implementations.

The floating point representation (`paFloat32`) uses +1.0 and -1.0 as the maximum and minimum respectively.

`paUInt8` is an unsigned 8 bit format where 128 is considered "ground"

The `paNonInterleaved` flag indicates that audio data is passed as an array of pointers to separate buffers, one buffer for each channel. Usually, when this flag is not used, audio data is passed as a single buffer with all channels interleaved.

Siehe auch

[Pa_OpenStream](#), [Pa_OpenDefaultStream](#), [PaDeviceInfo](#)
[paFloat32](#), [paInt16](#), [paInt32](#), [paInt24](#), [paInt8](#)
[paUInt8](#), [paCustomFormat](#), [paNonInterleaved](#)

6.1.3.9 typedef void PaStream

A single `PaStream` can provide multiple channels of real-time streaming audio input and output to a client application. A stream provides access to audio hardware represented by one or more `PaDevices`. Depending on the underlying Host API, it may be possible to open multiple streams using the same device, however this behavior is implementation defined. Portable applications should assume that a `PaDevice` may be simultaneously used by at most one `PaStream`.

Pointers to `PaStream` objects are passed between PortAudio functions that operate on streams.

Siehe auch

[Pa_OpenStream](#), [Pa_OpenDefaultStream](#), [Pa_OpenDefaultStream](#), [Pa_CloseStream](#), [Pa_StartStream](#), [Pa_StopStream](#), [Pa_AbortStream](#), [Pa_IsStreamActive](#), [Pa_GetStreamTime](#), [Pa_GetStreamCpuLoad](#)

6.1.3.10 typedef int PaStreamCallback(const void *input, void *output, unsigned long frameCount, const PaStreamCallbackTimeInfo *timeInfo, PaStreamCallbackFlags statusFlags, void *userData)

Functions of type `PaStreamCallback` are implemented by PortAudio clients. They consume, process or generate audio in response to requests from an active PortAudio stream.

When a stream is running, PortAudio calls the stream callback periodically. The callback function is responsible for processing buffers of audio samples passed via the input and output parameters.

The PortAudio stream callback runs at very high or real-time priority. It is required to consistently meet its time deadlines. Do not allocate memory, access the file system, call library functions or call other functions from the stream callback that may block or take an unpredictable amount of time to complete.

In order for a stream to maintain glitch-free operation the callback must consume and return audio data faster than it is recorded and/or played. PortAudio anticipates that each callback invocation may execute for a duration approaching the duration of `frameCount` audio frames at the stream sample rate. It is reasonable to expect to be able to utilise 70% or more of the available CPU time in the PortAudio callback. However, due to buffer size adaption and other factors, not all host APIs are able to guarantee audio stability under heavy CPU load with arbitrary fixed callback buffer sizes. When high callback CPU utilisation is required the most robust behavior can be achieved by using `paFramesPerBufferUnspecified` as the [Pa_OpenStream\(\)](#) `framesPerBuffer` parameter.

Parameter

<i>input</i>	and
<i>output</i>	are either arrays of interleaved samples or; if non-interleaved samples were requested using the <code>paNonInterleaved</code> sample format flag, an array of buffer pointers, one non-interleaved buffer for each channel.

The format, packing and number of channels used by the buffers are determined by parameters to [Pa_OpenStream\(\)](#).

Parameter

<i>frameCount</i>	The number of sample frames to be processed by the stream callback.
<i>timeInfo</i>	Timestamps indicating the ADC capture time of the first sample in the input buffer, the DAC output time of the first sample in the output buffer and the time the callback was invoked. See PaStreamCallbackTimeInfo and Pa_GetStreamTime()
<i>statusFlags</i>	Flags indicating whether input and/or output buffers have been inserted or will be dropped to overcome underflow or overflow conditions.
<i>userData</i>	The value of a user supplied pointer passed to Pa_OpenStream() intended for storing synthesis data etc.

Rückgabe

The stream callback should return one of the values in the [PaStreamCallbackResult](#) enumeration. To ensure that the callback continues to be called, it should return `paContinue` (0). Either `paComplete` or `paAbort` can be returned to finish stream processing, after either of these values is returned the callback will not be called again. If `paAbort` is returned the stream will finish as soon as possible. If `paComplete` is returned, the stream will continue until all buffers generated by the callback have been played. This may be useful in applications such as soundfile players where a specific duration of output is required. However, it is not necessary to utilize this mechanism as [Pa_StopStream\(\)](#), [Pa_AbortStream\(\)](#) or [Pa_CloseStream\(\)](#) can also be used to stop the stream. The callback must always fill the entire output buffer irrespective of its return value.

Siehe auch

[Pa_OpenStream](#), [Pa_OpenDefaultStream](#)

Zu beachten

With the exception of [Pa_GetStreamCpuLoad\(\)](#) it is not permissible to call PortAudio API functions from within the stream callback.

6.1.3.11 typedef unsigned long PaStreamCallbackFlags

Flag bit constants for the `statusFlags` to `PaStreamCallback`.

Siehe auch

[paInputUnderflow](#), [paInputOverflow](#), [paOutputUnderflow](#), [paOutputOverflow](#), [paPrimingOutput](#)

6.1.3.12 typedef enum PaStreamCallbackResult PaStreamCallbackResult

Allowable return values for the `PaStreamCallback`.

Siehe auch

[PaStreamCallback](#)

6.1.3.13 `typedef struct PaStreamCallbackTimeInfo PaStreamCallbackTimeInfo`

Timing information for the buffers passed to the stream callback.

Time values are expressed in seconds and are synchronised with the time base used by [Pa_GetStreamTime\(\)](#) for the associated stream.

Siehe auch

[PaStreamCallback](#), [Pa_GetStreamTime](#)

6.1.3.14 `typedef void PaStreamFinishedCallback(void *userData)`

Functions of type `PaStreamFinishedCallback` are implemented by PortAudio clients. They can be registered with a stream using the `Pa_SetStreamFinishedCallback` function. Once registered they are called when the stream becomes inactive (ie once a call to [Pa_StopStream\(\)](#) will not block). A stream will become inactive after the stream callback returns non-zero, or when `Pa_StopStream` or `Pa_AbortStream` is called. For a stream providing audio output, if the stream callback returns `paComplete`, or `Pa_StopStream` is called, the stream finished callback will not be called until all generated sample data has been played.

Parameter

<i>userData</i>	The <i>userData</i> parameter supplied to Pa_OpenStream()
-----------------	---

Siehe auch

[Pa_SetStreamFinishedCallback](#)

6.1.3.15 `typedef unsigned long PaStreamFlags`

Flags used to control the behavior of a stream. They are passed as parameters to `Pa_OpenStream` or `Pa_OpenDefaultStream`. Multiple flags may be ORed together.

Siehe auch

[Pa_OpenStream](#), [Pa_OpenDefaultStream](#)
[paNoFlag](#), [paClipOff](#), [paDitherOff](#), [paNeverDropInput](#), [paPrimeOutputBuffersUsingStreamCallback](#), [pa-PlatformSpecificFlags](#)

6.1.3.16 `typedef struct PaStreamInfo PaStreamInfo`

A structure containing unchanging information about an open stream.

Siehe auch

[Pa_GetStreamInfo](#)

6.1.3.17 `typedef struct PaStreamParameters PaStreamParameters`

Parameters for one direction (input or output) of a stream.

6.1.3.18 typedef double PaTime

The type used to represent monotonic time in seconds. PaTime is used for the fields of the [PaStreamCallbackTimeInfo](#) argument to the PaStreamCallback and as the result of [Pa_GetStreamTime\(\)](#).

PaTime values have unspecified origin.

Siehe auch

[PaStreamCallback](#), [PaStreamCallbackTimeInfo](#), [Pa_GetStreamTime](#)

6.1.4 Dokumentation der Aufzählungstypen

6.1.4.1 enum PaHostApiTypeId

Unchanging unique identifiers for each supported host API. This type is used in the [PaHostApiInfo](#) structure. The values are guaranteed to be unique and to never change, thus allowing code to be written that conditionally uses host API specific extensions.

New type ids will be allocated when support for a host API reaches "public alpha" status, prior to that developers should use the paInDevelopment type id.

Siehe auch

[PaHostApiInfo](#)

6.1.4.2 enum PaStreamCallbackResult

Allowable return values for the PaStreamCallback.

Siehe auch

[PaStreamCallback](#)

Aufzählungswerte

paContinue Signal that the stream should continue invoking the callback and processing audio.

paComplete Signal that the stream should stop invoking the callback and finish once all output samples have played.

paAbort Signal that the stream should stop invoking the callback and finish as soon as possible.

6.1.5 Dokumentation der Funktionen

6.1.5.1 PaError Pa_AbortStream (PaStream * *stream*)

Terminates audio processing immediately without waiting for pending buffers to complete.

6.1.5.2 PaError Pa_CloseStream (PaStream * *stream*)

Closes an audio stream. If the audio stream is active it discards any pending buffers as if [Pa_AbortStream\(\)](#) had been called.

6.1.5.3 PaHostApiIndex Pa_GetDefaultHostApi (void)

Retrieve the index of the default host API. The default host API will be the lowest common denominator host API on the current platform and is unlikely to provide the best performance.

Rückgabe

A non-negative value ranging from 0 to ([Pa_GetHostApiCount\(\)](#)-1) indicating the default host API index or, a [PaErrorCode](#) (which are always negative) if PortAudio is not initialized or an error is encountered.

6.1.5.4 [PaDeviceIndex](#) [Pa_GetDefaultInputDevice](#) (void)

Retrieve the index of the default input device. The result can be used in the `inputDevice` parameter to [Pa_OpenStream\(\)](#).

Rückgabe

The default input device index for the default host API, or `paNoDevice` if no default input device is available or an error was encountered.

6.1.5.5 [PaDeviceIndex](#) [Pa_GetDefaultOutputDevice](#) (void)

Retrieve the index of the default output device. The result can be used in the `outputDevice` parameter to [Pa_OpenStream\(\)](#).

Rückgabe

The default output device index for the default host API, or `paNoDevice` if no default output device is available or an error was encountered.

Zu beachten

On the PC, the user can specify a default device by setting an environment variable. For example, to use device #1.

```
set PA_RECOMMENDED_OUTPUT_DEVICE=1
```

The user should first determine the available device ids by using the supplied application "pa_devs".

6.1.5.6 [PaDeviceIndex](#) [Pa_GetDeviceCount](#) (void)

Retrieve the number of available devices. The number of available devices may be zero.

Rückgabe

A non-negative value indicating the number of available devices or, a [PaErrorCode](#) (which are always negative) if PortAudio is not initialized or an error is encountered.

6.1.5.7 [const PaDeviceInfo*](#) [Pa_GetDeviceInfo](#) ([PaDeviceIndex](#) *device*)

Retrieve a pointer to a [PaDeviceInfo](#) structure containing information about the specified device.

Rückgabe

A pointer to an immutable [PaDeviceInfo](#) structure. If the device parameter is out of range the function returns NULL.

Parameter

<i>device</i>	A valid device index in the range 0 to (Pa_GetDeviceCount() -1)
---------------	--

Zu beachten

PortAudio manages the memory referenced by the returned pointer, the client must not manipulate or free the memory. The pointer is only guaranteed to be valid between calls to [Pa_Initialize\(\)](#) and [Pa_Terminate\(\)](#).

Siehe auch

[PaDeviceInfo](#), [PaDeviceIndex](#)

6.1.5.8 `const char* Pa_GetErrorText (PaError errorCode)`

Translate the supplied PortAudio error code into a human readable message.

6.1.5.9 `PaHostApiIndex Pa_GetHostApiCount (void)`

Retrieve the number of available host APIs. Even if a host API is available it may have no devices available.

Rückgabe

A non-negative value indicating the number of available host APIs or, a `PaErrorCode` (which are always negative) if PortAudio is not initialized or an error is encountered.

Siehe auch

[PaHostApiIndex](#)

6.1.5.10 `const PaHostApiInfo* Pa_GetHostApiInfo (PaHostApiIndex hostApi)`

Retrieve a pointer to a structure containing information about a specific host Api.

Parameter

<i>hostApi</i>	A valid host API index ranging from 0 to (Pa_GetHostApiCount() -1)
----------------	---

Rückgabe

A pointer to an immutable [PaHostApiInfo](#) structure describing a specific host API. If the `hostApi` parameter is out of range or an error is encountered, the function returns NULL.

The returned structure is owned by the PortAudio implementation and must not be manipulated or freed. The pointer is only guaranteed to be valid between calls to [Pa_Initialize\(\)](#) and [Pa_Terminate\(\)](#).

6.1.5.11 `const PaHostErrorInfo* Pa_GetLastHostErrorInfo (void)`

Return information about the last host error encountered. The error information returned by [Pa_GetLastHostErrorInfo\(\)](#) will never be modified asynchronously by errors occurring in other PortAudio owned threads (such as the thread that manages the stream callback.)

This function is provided as a last resort, primarily to enhance debugging by providing clients with access to all available error information.

Rückgabe

A pointer to an immutable structure constraining information about the host error. The values in this structure will only be valid if a PortAudio function has previously returned the `paUnanticipatedHostError` error code.

6.1.5.12 PaError Pa_GetSampleSize (PaSampleFormat *format*)

Retrieve the size of a given sample format in bytes.

Rückgabe

The size in bytes of a single sample in the specified format, or `paSampleFormatNotSupported` if the format is not supported.

6.1.5.13 double Pa_GetStreamCpuLoad (PaStream * *stream*)

Retrieve CPU usage information for the specified stream. The "CPU Load" is a fraction of total CPU time consumed by a callback stream's audio processing routines including, but not limited to the client supplied stream callback. This function does not work with blocking read/write streams.

This function may be called from the stream callback function or the application.

Rückgabe

A floating point value, typically between 0.0 and 1.0, where 1.0 indicates that the stream callback is consuming the maximum number of CPU cycles possible to maintain real-time operation. A value of 0.5 would imply that PortAudio and the stream callback was consuming roughly 50% of the available CPU time. The return value may exceed 1.0. A value of 0.0 will always be returned for a blocking read/write stream, or if an error occurs.

6.1.5.14 PaHostApiTypeId Pa_GetStreamHostApiType (PaStream * *stream*)

Retrieve the host type handling an open stream.

Rückgabe

Returns a non-negative value representing the host API type handling an open stream or, a `PaErrorCode` (which are always negative) if PortAudio is not initialized or an error is encountered.

6.1.5.15 const PaStreamInfo* Pa_GetStreamInfo (PaStream * *stream*)

Retrieve a pointer to a [PaStreamInfo](#) structure containing information about the specified stream.

Rückgabe

A pointer to an immutable [PaStreamInfo](#) structure. If the stream parameter invalid, or an error is encountered, the function returns NULL.

Parameter

<i>stream</i>	A pointer to an open stream previously created with <code>Pa_OpenStream</code> .
---------------	--

Zu beachten

PortAudio manages the memory referenced by the returned pointer, the client must not manipulate or free the memory. The pointer is only guaranteed to be valid until the specified stream is closed.

Siehe auch

[PaStreamInfo](#)

6.1.5.16 signed long Pa_GetStreamReadAvailable (PaStream * stream)

Retrieve the number of frames that can be read from the stream without waiting.

Rückgabe

Returns a non-negative value representing the maximum number of frames that can be read from the stream without blocking or busy waiting or, a PaErrorCode (which are always negative) if PortAudio is not initialized or an error is encountered.

6.1.5.17 PaTime Pa_GetStreamTime (PaStream * stream)

Returns the current time in seconds for a stream according to the same clock used to generate callback [PaStreamCallbackTimeInfo](#) timestamps. The time values are monotonically increasing and have unspecified origin.

Pa_GetStreamTime returns valid time values for the entire life of the stream, from when the stream is opened until it is closed. Starting and stopping the stream does not affect the passage of time returned by Pa_GetStreamTime.

This time may be used for synchronizing other events to the audio stream, for example synchronizing audio to MIDI.

Rückgabe

The stream's current time in seconds, or 0 if an error occurred.

Siehe auch

[PaTime](#), [PaStreamCallback](#), [PaStreamCallbackTimeInfo](#)

6.1.5.18 signed long Pa_GetStreamWriteAvailable (PaStream * stream)

Retrieve the number of frames that can be written to the stream without waiting.

Rückgabe

Returns a non-negative value representing the maximum number of frames that can be written to the stream without blocking or busy waiting or, a PaErrorCode (which are always negative) if PortAudio is not initialized or an error is encountered.

6.1.5.19 int Pa_GetVersion (void)

Retrieve the release number of the currently running PortAudio build, eg 1900.

6.1.5.20 const char* Pa_GetVersionText (void)

Retrieve a textual description of the current PortAudio build, eg "PortAudio V19-devel 13 October 2002".

6.1.5.21 PaDeviceIndex Pa_HostApiDeviceIndexToDeviceIndex (PaHostApiIndex hostApi, int hostApiDeviceIndex)

Convert a host-API-specific device index to standard PortAudio device index. This function may be used in conjunction with the deviceCount field of [PaHostApiInfo](#) to enumerate all devices for the specified host API.

Parameter

<i>hostApi</i>	A valid host API index ranging from 0 to (Pa_GetHostApiCount())-1)
<i>hostApiDevice-Index</i>	A valid per-host device index in the range 0 to (Pa_GetHostApiInfo(hostApi)->deviceCount -1)

Rückgabe

A non-negative `PaDeviceIndex` ranging from 0 to ([Pa_GetDeviceCount\(\)](#))-1) or, a `PaErrorCode` (which are always negative) if PortAudio is not initialized or an error is encountered.

A `paInvalidHostApi` error code indicates that the host API index specified by the `hostApi` parameter is out of range.

A `paInvalidDevice` error code indicates that the `hostApiDeviceIndex` parameter is out of range.

Siehe auch

[PaHostApiInfo](#)

6.1.5.22 `PaHostApiIndex Pa_HostApiTypeToHostApiIndex (PaHostApiType type)`

Convert a static host API unique identifier, into a runtime host API index.

Parameter

<i>type</i>	A unique host API identifier belonging to the <code>PaHostApiType</code> enumeration.
-------------	---

Rückgabe

A valid `PaHostApiIndex` ranging from 0 to ([Pa_GetHostApiCount\(\)](#))-1) or, a `PaErrorCode` (which are always negative) if PortAudio is not initialized or an error is encountered.

The `paHostApiNotFound` error code indicates that the host API specified by the `type` parameter is not available.

Siehe auch

[PaHostApiType](#)

6.1.5.23 `PaError Pa_Initialize (void)`

Library initialization function - call this before using PortAudio. This function initializes internal data structures and prepares underlying host APIs for use. With the exception of [Pa_GetVersion\(\)](#), [Pa_GetVersionText\(\)](#), and [Pa_GetErrorText\(\)](#), this function MUST be called before using any other PortAudio API functions.

If [Pa_Initialize\(\)](#) is called multiple times, each successful call must be matched with a corresponding call to [Pa_Terminate\(\)](#). Pairs of calls to [Pa_Initialize\(\)](#)/[Pa_Terminate\(\)](#) may overlap, and are not required to be fully nested.

Note that if [Pa_Initialize\(\)](#) returns an error code, [Pa_Terminate\(\)](#) should NOT be called.

Rückgabe

`paNoError` if successful, otherwise an error code indicating the cause of failure.

Siehe auch

[Pa_Terminate](#)

6.1.5.24 `PaError Pa_IsFormatSupported (const PaStreamParameters * inputParameters, const PaStreamParameters * outputParameters, double sampleRate)`

Determine whether it would be possible to open a stream with the specified parameters.

Parameter

<i>inputParameters</i>	A structure that describes the input parameters used to open a stream. The suggested-Latency field is ignored. See PaStreamParameters for a description of these parameters. inputParameters must be NULL for output-only streams.
<i>outputParameters</i>	A structure that describes the output parameters used to open a stream. The suggested-Latency field is ignored. See PaStreamParameters for a description of these parameters. outputParameters must be NULL for input-only streams.
<i>sampleRate</i>	The required sampleRate. For full-duplex streams it is the sample rate for both input and output

Rückgabe

Returns 0 if the format is supported, and an error code indicating why the format is not supported otherwise. The constant `paFormatIsSupported` is provided to compare with the return value for success.

Siehe auch

[paFormatIsSupported](#), [PaStreamParameters](#)

6.1.5.25 **PaError Pa_IsStreamActive (PaStream * stream)**

Determine whether the stream is active. A stream is active after a successful call to [Pa_StartStream\(\)](#), until it becomes inactive either as a result of a call to [Pa_StopStream\(\)](#) or [Pa_AbortStream\(\)](#), or as a result of a return value other than `paContinue` from the stream callback. In the latter case, the stream is considered inactive after the last buffer has finished playing.

Rückgabe

Returns one (1) when the stream is active (ie playing or recording audio), zero (0) when not playing or, a `PaErrorCode` (which are always negative) if PortAudio is not initialized or an error is encountered.

Siehe auch

[Pa_StopStream](#), [Pa_AbortStream](#), [Pa_IsStreamStopped](#)

6.1.5.26 **PaError Pa_IsStreamStopped (PaStream * stream)**

Determine whether the stream is stopped. A stream is considered to be stopped prior to a successful call to [Pa_StartStream](#) and after a successful call to [Pa_StopStream](#) or [Pa_AbortStream](#). If a stream callback returns a value other than `paContinue` the stream is NOT considered to be stopped.

Rückgabe

Returns one (1) when the stream is stopped, zero (0) when the stream is running or, a `PaErrorCode` (which are always negative) if PortAudio is not initialized or an error is encountered.

Siehe auch

[Pa_StopStream](#), [Pa_AbortStream](#), [Pa_IsStreamActive](#)

6.1.5.27 **PaError Pa_OpenDefaultStream (PaStream ** stream, int numInputChannels, int numOutputChannels, PaSampleFormat sampleFormat, double sampleRate, unsigned long framesPerBuffer, PaStreamCallback * streamCallback, void * userData)**

A simplified version of [Pa_OpenStream\(\)](#) that opens the default input and/or output devices.

Parameter

<i>stream</i>	The address of a PaStream pointer which will receive a pointer to the newly opened stream.
<i>numInput-Channels</i>	The number of channels of sound that will be supplied to the stream callback or returned by Pa_ReadStream. It can range from 1 to the value of maxInputChannels in the PaDeviceInfo record for the default input device. If 0 the stream is opened as an output-only stream.
<i>numOutput-Channels</i>	The number of channels of sound to be delivered to the stream callback or passed to Pa_WriteStream. It can range from 1 to the value of maxOutputChannels in the PaDeviceInfo record for the default output device. If 0 the stream is opened as an output-only stream.
<i>sampleFormat</i>	The sample format of both the input and output buffers provided to the callback or passed to and from Pa_ReadStream and Pa_WriteStream. sampleFormat may be any of the formats described by the PaSampleFormat enumeration.
<i>sampleRate</i>	Same as Pa_OpenStream parameter of the same name.
<i>framesPerBuffer</i>	Same as Pa_OpenStream parameter of the same name.
<i>streamCallback</i>	Same as Pa_OpenStream parameter of the same name.
<i>userData</i>	Same as Pa_OpenStream parameter of the same name.

Rückgabe

As for Pa_OpenStream

Siehe auch

[Pa_OpenStream](#), [PaStreamCallback](#)

6.1.5.28 PaError Pa_OpenStream (PaStream ** stream, const PaStreamParameters * inputParameters, const PaStreamParameters * outputParameters, double sampleRate, unsigned long framesPerBuffer, PaStreamFlags streamFlags, PaStreamCallback * streamCallback, void * userData)

Opens a stream for either input, output or both.

Parameter

<i>stream</i>	The address of a PaStream pointer which will receive a pointer to the newly opened stream.
<i>inputParameters</i>	A structure that describes the input parameters used by the opened stream. See PaStreamParameters for a description of these parameters. inputParameters must be NULL for output-only streams.
<i>output-Parameters</i>	A structure that describes the output parameters used by the opened stream. See PaStreamParameters for a description of these parameters. outputParameters must be NULL for input-only streams.
<i>sampleRate</i>	The desired sampleRate. For full-duplex streams it is the sample rate for both input and output
<i>framesPerBuffer</i>	The number of frames passed to the stream callback function, or the preferred block granularity for a blocking read/write stream. The special value paFramesPerBufferUnspecified (0) may be used to request that the stream callback will receive an optimal (and possibly varying) number of frames based on host requirements and the requested latency settings. Note: With some host APIs, the use of non-zero framesPerBuffer for a callback stream may introduce an additional layer of buffering which could introduce additional latency. PortAudio guarantees that the additional latency will be kept to the theoretical minimum however, it is strongly recommended that a non-zero framesPerBuffer value only be used when your algorithm requires a fixed number of frames per stream callback.

<i>streamFlags</i>	Flags which modify the behavior of the streaming process. This parameter may contain a combination of flags ORed together. Some flags may only be relevant to certain buffer formats.
<i>streamCallback</i>	A pointer to a client supplied function that is responsible for processing and filling input and output buffers. If this parameter is NULL the stream will be opened in 'blocking read/write' mode. In blocking mode, the client can receive sample data using <code>Pa_ReadStream</code> and write sample data using <code>Pa_WriteStream</code> , the number of samples that may be read or written without blocking is returned by <code>Pa_GetStreamReadAvailable</code> and <code>Pa_GetStreamWriteAvailable</code> respectively.
<i>userData</i>	A client supplied pointer which is passed to the stream callback function. It could for example, contain a pointer to instance data necessary for processing the audio buffers. This parameter is ignored if <code>streamCallback</code> is NULL.

Rückgabe

Upon success `Pa_OpenStream()` returns `paNoError` and places a pointer to a valid `PaStream` in the stream argument. The stream is inactive (stopped). If a call to `Pa_OpenStream()` fails, a non-zero error code is returned (see `PaError` for possible error codes) and the value of stream is invalid.

Siehe auch

[PaStreamParameters](#), [PaStreamCallback](#), [Pa_ReadStream](#), [Pa_WriteStream](#), [Pa_GetStreamReadAvailable](#), [Pa_GetStreamWriteAvailable](#)

6.1.5.29 `PaError Pa_ReadStream (PaStream * stream, void * buffer, unsigned long frames)`

Read samples from an input stream. The function doesn't return until the entire buffer has been filled - this may involve waiting for the operating system to supply the data.

Parameter

<i>stream</i>	A pointer to an open stream previously created with <code>Pa_OpenStream</code> .
<i>buffer</i>	A pointer to a buffer of sample frames. The buffer contains samples in the format specified by the <code>inputParameters->sampleFormat</code> field used to open the stream, and the number of channels specified by <code>inputParameters->numChannels</code> . If non-interleaved samples were requested using the <code>paNonInterleaved</code> sample format flag, buffer is a pointer to the first element of an array of buffer pointers, one non-interleaved buffer for each channel.
<i>frames</i>	The number of frames to be read into buffer. This parameter is not constrained to a specific range, however high performance applications will want to match this parameter to the <code>framesPerBuffer</code> parameter used when opening the stream.

Rückgabe

On success `PaNoError` will be returned, or `PaInputOverflowed` if input data was discarded by PortAudio after the previous call and before this call.

6.1.5.30 `PaError Pa_SetStreamFinishedCallback (PaStream * stream, PaStreamFinishedCallback * streamFinishedCallback)`

Register a stream finished callback function which will be called when the stream becomes inactive. See the description of `PaStreamFinishedCallback` for further details about when the callback will be called.

Parameter

<i>stream</i>	a pointer to a PaStream that is in the stopped state - if the stream is not stopped, the stream's finished callback will remain unchanged and an error code will be returned.
<i>streamFinished-Callback</i>	a pointer to a function with the same signature as PaStreamFinishedCallback, that will be called when the stream becomes inactive. Passing NULL for this parameter will un-register a previously registered stream finished callback function.

Rückgabe

on success returns paNoError, otherwise an error code indicating the cause of the error.

Siehe auch

[PaStreamFinishedCallback](#)

6.1.5.31 void Pa_Sleep (long msec)

Put the caller to sleep for at least 'msec' milliseconds. This function is provided only as a convenience for authors of portable code (such as the tests and examples in the PortAudio distribution.)

The function may sleep longer than requested so don't rely on this for accurate musical timing.

6.1.5.32 PaError Pa_StartStream (PaStream * stream)

Commences audio processing.

6.1.5.33 PaError Pa_StopStream (PaStream * stream)

Terminates audio processing. It waits until all pending audio buffers have been played before it returns.

6.1.5.34 PaError Pa_Terminate (void)

Library termination function - call this when finished using PortAudio. This function deallocates all resources allocated by PortAudio since it was initialized by a call to [Pa_Initialize\(\)](#). In cases where [Pa_Initialise\(\)](#) has been called multiple times, each call must be matched with a corresponding call to [Pa_Terminate\(\)](#). The final matching call to [Pa_Terminate\(\)](#) will automatically close any PortAudio streams that are still open.

[Pa_Terminate\(\)](#) MUST be called before exiting a program which uses PortAudio. Failure to do so may result in serious resource leaks, such as audio devices not being available until the next reboot.

Rückgabe

paNoError if successful, otherwise an error code indicating the cause of failure.

Siehe auch

[Pa_Initialize](#)

6.1.5.35 PaError Pa_WriteStream (PaStream * stream, const void * buffer, unsigned long frames)

Write samples to an output stream. This function doesn't return until the entire buffer has been consumed - this may involve waiting for the operating system to consume the data.

Parameter

<i>stream</i>	A pointer to an open stream previously created with Pa_OpenStream.
<i>buffer</i>	A pointer to a buffer of sample frames. The buffer contains samples in the format specified by the outputParameters->sampleFormat field used to open the stream, and the number of channels specified by outputParameters->numChannels. If non-interleaved samples were requested using the paNonInterleaved sample format flag, buffer is a pointer to the first element of an array of buffer pointers, one non-interleaved buffer for each channel.
<i>frames</i>	The number of frames to be written from buffer. This parameter is not constrained to a specific range, however high performance applications will want to match this parameter to the framesPerBuffer parameter used when opening the stream.

Rückgabe

On success PaNoError will be returned, or paOutputUnderflowed if additional output data was inserted after the previous call and before this call.