Blokus

Entwicklerdokumentation

Frank Stolze
Alex Besstschastnich
Sascha Hlusiak

22.06.05

Inhaltsverzeichnis

1.Installationsprogramm

Das Installationsprogramm wurde mit dem kostenlosen Compiler der Firma Nullsoft (NSIS) erstellt. Es musste zuerst ein Script (*.nsi) geschrieben werden, welches der Compiler dann übersetzen konnte. Das Programm bietet dem Benutzer die Möglichkeit die Installation auf englisch oder deutsch zu starten. Weiter werden vom Benutzer die zu installierenden Komponenten ausgewählt. Nach der Angabe eines Pfades, wird gefragt, ob ein Start Menü Eintrag gemacht werden soll. Zum Schluss werden die Daten von der Cd auf die Festplatte kopiert.
Beim Installieren wird auch sofort ein „uninstaller“ kreiert, der dann die kopierten Dateien wieder entfernt.
Da bei der Installation gleichzeitig ein Eintrag im Registerverzeichnis gemacht wird, ist es ohne weiteres möglich das Spiel auch über Systemsteuerung -> Software zu deinstallieren.

2.Autostart Programm

Damit ein Programm überhaupt automatisch von der Cd starten kann, muss vorerst ein kleines Skript geschrieben werde. Dieses Skript muss *autorun.inf* heißen, damit es von Windows erkannt wird. Der Inhalt der inf-Datei sieht so aus:

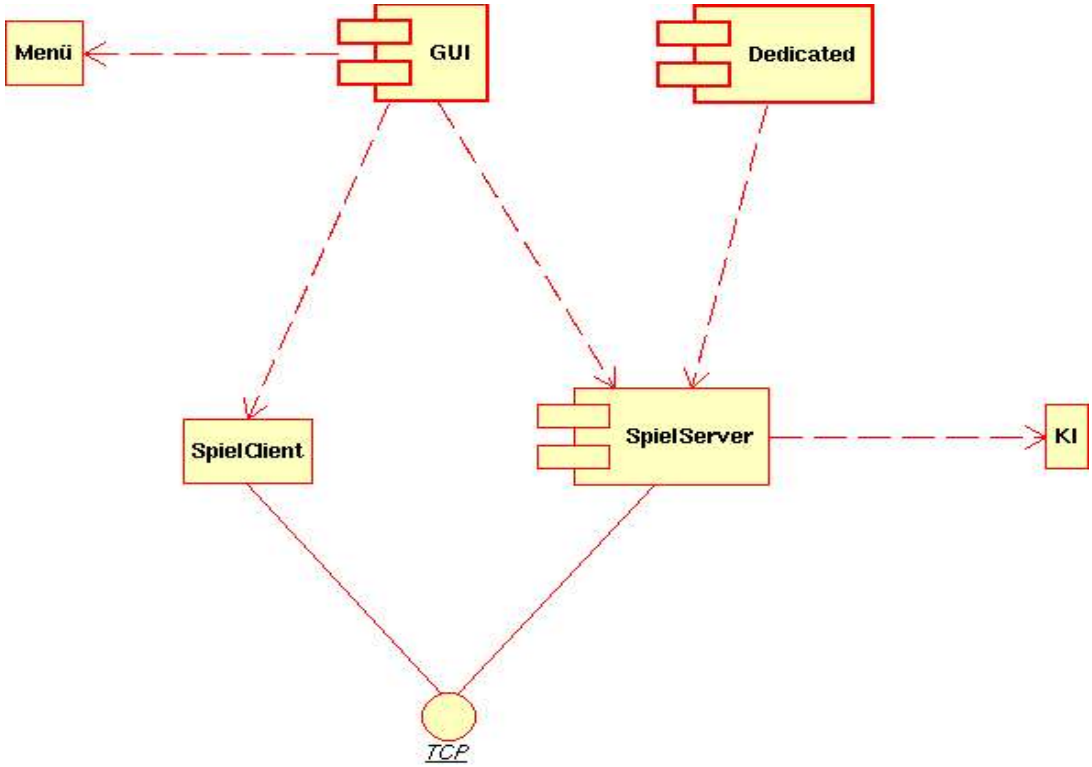
```
[autorun]
ICON=BLOKUS.ICO
OPEN=Start.EXE
```

Die erste Zeile dient als Kennung, die zweite Zeile definiert das Icon, welches beim Einlegen der Cd angezeigt werden soll. Zeile drei gibt an, welche Datei ausgeführt werden soll.

Mit *Start.exe* wird noch nicht das Spiel Blokus gestartet, sondern vorerst ein anderes Programm, das den Blokus-Navigator darstellt, und den Benutzer helfen soll die vorhandenen Optionen sinnvoll zu nutzen. Einige der Optionen sind z.B. „Blokus spielen“ oder „Blokus installieren“ (siehe auch Benutzerdokumentation). Dieses Programm wurde mit dem C++Builder 6.0 erstellt und ist im Wesentlichen nur ein Formular, das fünf Knöpfe enthält. Jeder Knopf startet ein weiteres Programm. Ein Nachteil dieser Lösung ist, das zum Starten des, mit dem C++Builder geschriebenen Programms, einige dll-Dateien mit eingebunden werden müssen: borlndmm.dll, cc3260mt.dll, rtl60.bpl, vcl60.bpl. Diese ermöglichen das Starten auf jedem Windows Computer.

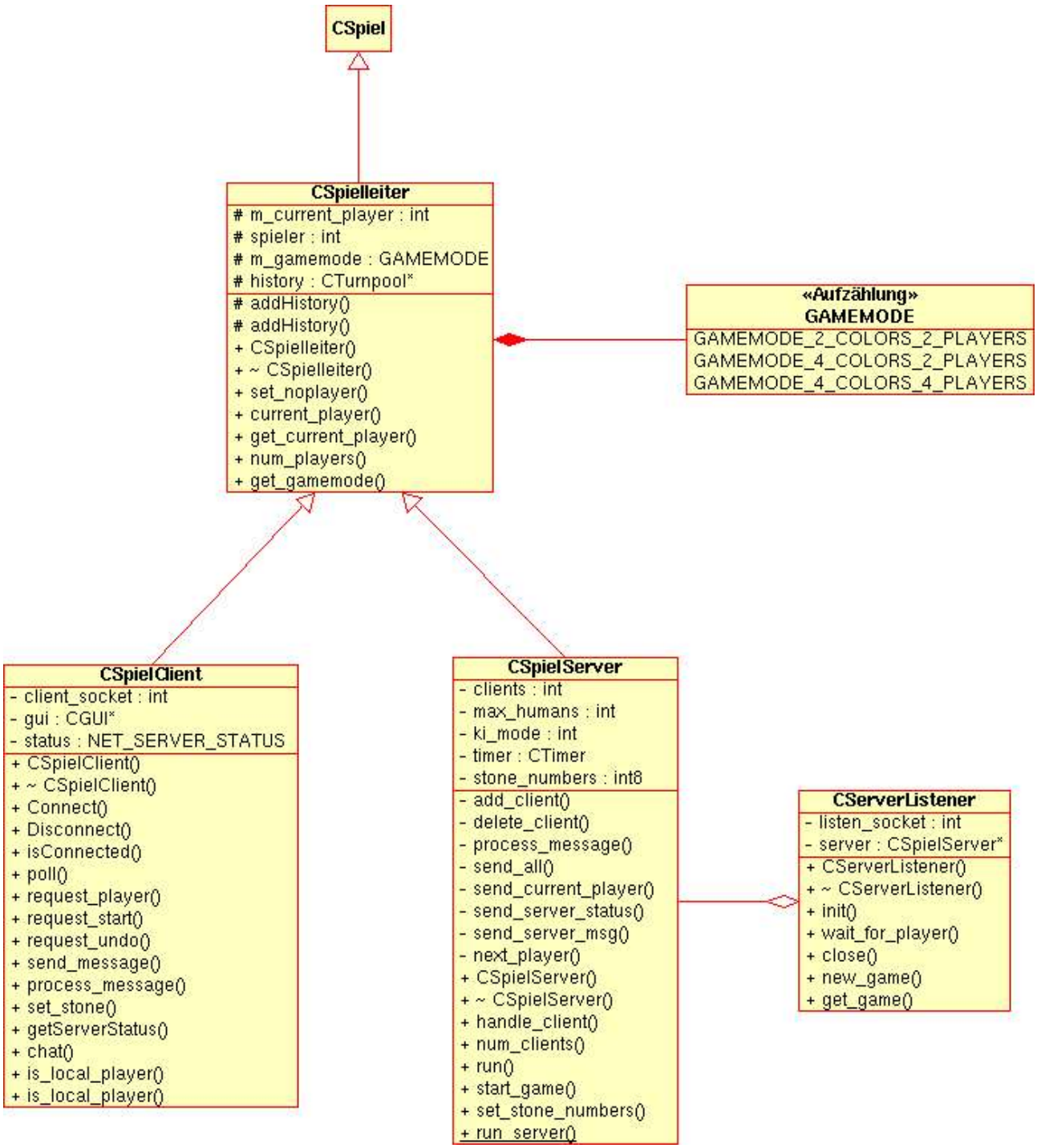
3.Komponenten des Spiels

Das Spiel lässt sich grob in folgende Teilbereiche untergliedern:



- GUI stellt das Hauptprogramm bereit, sie stellt das Spielfeld grafisch dar und bietet Interaktion mit dem Anwender. Die GUI beinhaltet SpielClient und SpielServer.
- Menü ist die Menüstruktur, die dem Benutzer genauere Einstellungen und Funktionen ermöglicht.
- Die Komponente SpielServer stellt den Spielleiter dar, sie verwaltet ein Spiel, beinhaltet großen Teil der Spiellogik und die gesamte KI, ist also für die Züge der Computergegner zuständig.
- Der SpielClient findet sich in der GUI wieder. Über diesen kommuniziert die GUI mit dem Spielleiter, setzt dort Züge ab und empfängt Züge anderer Mitspieler.
- SpielClient und SpielServer kommunizieren über TCP/IP.
- Es gibt einen Stand-Alone-Server, der nur den SpielServer mit der KI beinhaltet und ein Mehrspielerspiel bereitstellen kann.

4.Netzwerk


CSpiel
CSpielleiter
m_current_player : int
spieler : int
m_gamemode : GAMEMODE
history : CTurnpool*
addHistory()
addHistory()
+ CSpielleiter()
+ ~ CSpielleiter()
+ set_noplayer()
+ current_player()
+ get_current_player()
+ num_players()
+ get_gamemode()
«Aufzählung»
GAMEMODE
GAMEMODE_2_COLORS_2_PLAYERS
GAMEMODE_4_COLORS_2_PLAYERS
GAMEMODE_4_COLORS_4_PLAYERS
CSpielClient
- client_socket : int
- gui : CGUI*
- status : NET_SERVER_STATUS
+ CSpielClient()
+ ~ CSpielClient()
+ Connect()
+ Disconnect()
+ isConnected()
+ poll()
+ request_player()
+ request_start()
+ request_undo()
+ send_message()
+ process_message()
+ set_stone()
+ getServerStatus()
+ chat()
+ is_local_player()
+ is_local_player()
CSpielServer
- clients : int
- max_humans : int
- ki_mode : int
- timer : CTimer
- stone_numbers : int8
- add_client()
- delete_client()
- process_message()
- send_all()
- send_current_player()
- send_server_status()
- send_server_msg()
- next_player()
+ CSpielServer()
+ ~ CSpielServer()
+ handle_client()
+ num_clients()
+ run()
+ start_game()
+ set_stone_numbers()
+ run_server()
CServerListener
- listen_socket : int
- server : CSpielServer*
+ CServerListener()
+ ~ CServerListener()
+ init()
+ wait_for_player()
+ close()
+ new_game()
+ get_game()

Die Klasse CSpiel bietet Methoden und Variablen für ein Spiel, dazu mehr in der KI.

5.CSpielLeiter (spielleiter.cpp)

Der SpielLeiter erweitert die Klasse CSpiel um
- den aktuellen Spieler
- die Variable spieler, die über den Typ eines Spielers informiert (Lokal, Computer, Sonstige)
- den Spielmodus (4 Farben 4 Spieler, 4 Farben 2 Spieler und 2 Farben 2 Spieler)
- die Zug-History, um Züge rückgängig machen zu können.

CServerListener (serverlistener.cpp)

Der ServerListener hat folgende Aufgaben
- Ein TCP Socket erstellen, das Verbindungen akzeptiert.
- Einen CSpielServer instantiieren.
- Ankommende Verbindungen dem CSpielServer als Client hinzufügen.
- Sobald das Spiel gestartet wurde, soll der Listener keine ankommenden Verbindungen mehr akzeptieren.

CSpielServer (spielserver.cpp)

Der Server erbt von CSpielListener und CSpiel, beinhaltet alle Funktionen, um ein Spiel zu verwalten und erweitert diese um Netzwerkfunktionalität.
- Der Server speichert TCP Verbindungen zu allen verbundenen CSpielClients. Die freien Spieler werden durch Computerspieler aufgefüllt.
- Der Server wartet normalerweise auf Daten von den Clients, wertet sie aus und schickt sie an alle verbundenen Clients weiter.
- Er informiert die verbundenen Clients über den aktuellen Spieler.
- Clients schicken Züge an den Server. Der Server führt den Zug aus, und schickt ihn an alle Clients.
- Nach jedem Zug wählt der Server den nächsten möglichen Spieler und informiert Clients darüber. Besitzt kein Client den Spieler, führt der Server einen Zug für die KI aus und schickt den errechneten Zug an die Clients.

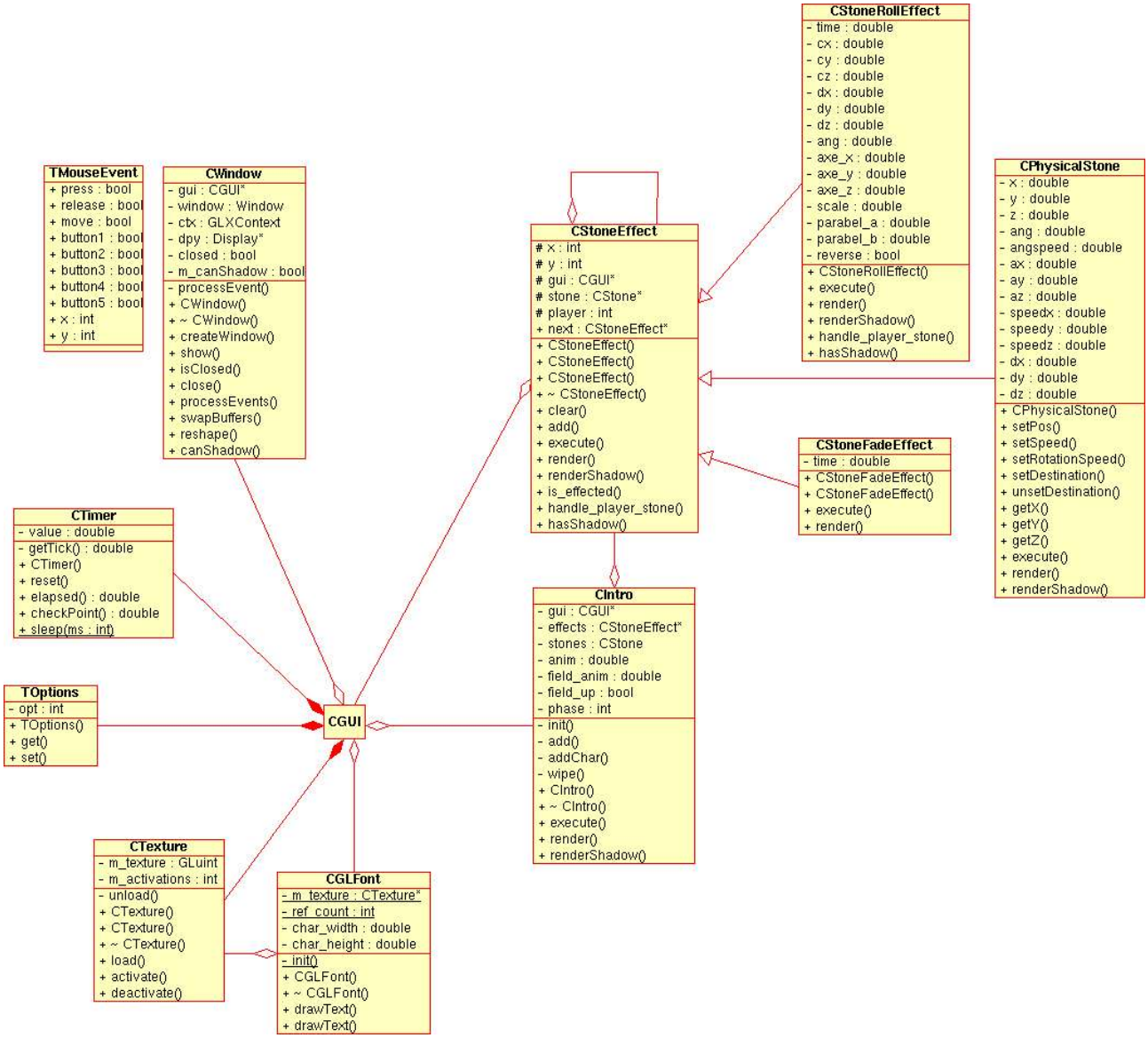
CSpielClient (spielclient.cpp)

Der Client hat direkte Verbindung zur GUI, nimmt von dieser Aktionen entgegen und informiert sie über Änderungen am Spiel.
- Der Client bietet Funktionen zum Aufbauen einer TCP Verbindung zu einem Server.
- Der SpielClient weiß, welcher Spieler dran ist, und ob dieser Spieler lokal von der GUI bearbeitet wird. Der Client fordert zu Beginn des Spiels lokale Spieler an, und merkt sich, welche er zugeteilt bekommt.

- Für lokale Spieler lässt die GUI Züge zu. Wird ein Zug getätigt, schickt der Client eine Zuganfrage an den SpielServer. Der Server führt ihn aus und informiert bei Erfolg alle Clients über den Zug. Erst dann wird der Zug vom SpielClient ausgeführt.
- Er sammelt Informationen über den Server und bietet Schnittstellen zum Versenden von z.B. Chat-Nachrichten.
- Da CSpielClient von CSpiel abstammt, beinhaltet der Client stets eine lokale Kopie vom Spielstand des Servers.
- Die GUI „pollt“ den Client in periodischen Abständen auf neue Nachrichten.

Für eine Liste der Netzwerknachrichten, die zwischen Server und Client ausgetauscht werden, bitte Blick in Datei „network.h“ und „network.cpp“ werfen.

6.GUI


CStoneRollEffect
- time : double
- cx : double
- cy : double
- cz : double
- dx : double
- dy : double
- dz : double
- ang : double
- axe_x : double
- axe_y : double
- axe_z : double
- scale : double
- parabel_a : double
- parabel_b : double
- reverse : bool
+ CStoneRollEffect()
+ execute()
+ render()
+ renderShadow()
+ handle_player_stone()
+ hasShadow()
CPhysicalStone
- x : double
- y : double
- z : double
- ang : double
- angspeed : double
- ax : double
- ay : double
- az : double
- speedx : double
- speedy : double
- speedz : double
- dx : double
- dy : double
- dz : double
+ CPhysicalStone()
+ setPos()
+ setSpeed()
+ setRotationSpeed()
+ setDestination()
+ unsetDestination()
+ getX()
+ getY()
+ getZ()
+ execute()
+ render()
+ renderShadow()
TMouseEvent
+ press : bool
+ release : bool
+ move : bool
+ button1 : bool
+ button2 : bool
+ button3 : bool
+ button4 : bool
+ button5 : bool
+ x : int
+ y : int
CWindow
- gui : CGUI*
- window : Window
- ctx : GLXContext
- dpy : Display*
- closed : bool
- m_canShadow : bool
- processEvent()
+ CWindow()
+ ~ CWindow()
+ createWindow()
+ show()
+ isClosed()
+ close()
+ processEvents()
+ swapBuffers()
+ reshape()
+ canShadow()
CStoneEffect
x : int
y : int
gui : CGUI*
stone : CStone*
player : int
+ next : CStoneEffect*
+ CStoneEffect()
+ CStoneEffect()
+ CStoneEffect()
+ ~ CStoneEffect()
+ clear()
+ add()
+ execute()
+ render()
+ renderShadow()
+ is_effected()
+ handle_player_stone()
+ hasShadow()
CStoneFadeEffect
- time : double
+ CStoneFadeEffect()
+ CStoneFadeEffect()
+ execute()
+ render()
CTimer
- value : double
- getTick() : double
+ CTimer()
+ reset()
+ elapsed() : double
+ checkPoint() : double
+ sleep(ms : int)
CIntro
- gui : CGUI*
- effects : CStoneEffect*
- stones : CStone
- anim : double
- field_anim : double
- field_up : bool
- phase : int
- init()
- add()
- addChar()
- wipe()
+ CIntro()
+ ~ CIntro()
+ execute()
+ render()
+ renderShadow()
TOptions
- opt : int
+ TOptions()
+ get()
+ set()
CGUI
CTexture
- m_texture : GLuint
- m_activations : int
- unload()
+ CTexture()
+ CTexture()
+ ~ CTexture()
+ load()
+ activate()
+ deactivate()
CGLFont
- m_texture : CTexture*
- ref_count : int
- char_width : double
- char_height : double
- init()
+ CGLFont()
+ ~ CGLFont()
+ drawText()
+ drawText()

CWindow (window.cpp)

Alles beginnt mit der Klasse CWindow (window.cpp). Sie stellt eine Kapselung eines Fensters auf dem Bildschirm dar, bietet also eine betriebssystemunabhängige Schnittstelle. Sie initialisiert das OpenGL System, es existieren derzeit zwei Implementationen davon, eine für Windows, die andere für X11 unter Linux/Unix. Als einheitliche Schnittstelle von Mausereignissen dient die Struktur TMouseEvent.

CTimer (timer.cpp)

Kapselung von Zeitfunktionen. Bietet Zugriff auf sehr genaue Timer für Zeitmessungen im Mikrosekundenbereich.

CTexture (texture.cpp)

Kapselt eine Textur, also ein 2D Bild für OpenGL und bietet so eine leicht zu benutzende Schnittstelle für die GUI, um die Texturen für Wand, Spielfeld und Tisch zu verwalten.

CGLFont (glfont.cpp)

Klasse zum Rendern von Text. Dazu existiert eine globale CTextur, mit allen Buchstaben als Schwarz/Weiß Informationen. Die Klasse bietet nun einfache Methoden zum Rendern einzelner Buchstaben und ganzer Zeichenketten in variabler Größe und Position.

CStoneEffect, CStoneFadeEffect, CStoneRollEffect, CPhysicalStone (stoneeffect.cpp)

CStoneEffect ist eine rekursive abstrakte Elternklasse für Effekte von Steinen. Als verkettete Liste implementiert, lassen sich nun beliebig viele Instanzen von Effekten hinzufügen.
CStoneFadeEffect wird von der GUI als Hinweis benutzt. Fordert der Spieler einen Hinweis an, erscheint ein blinkender CStoneFadeEffect auf dem Spielfeld.
CStoneRollEffect implementiert die Animation, die erscheint, wenn ein Spieler einen Stein auf das Spielfeld legt. Wird auch verwendet, wenn durch ein „Zug Rückgängig“, die Steine zurück zu den Spielern fliegen.
CPhysicalStone wird nur von CIntro benutzt, um die Steine durch die Luft fliegen zu lassen.

Die GUI rendert einen Stein auf dem Spielfeld nur, wenn das Feld nicht zu einem existierenden CStoneEffect gehört. Ist ein Effekt vorbei, wird er automatisch aus der verketteten Liste ausgehängt. CGUI und CIntro haben eigene CStoneEffect-Listen.

CIntro (intro.cpp)

Die Klasse implementiert die Intro-Sequenz. Dazu legt sie eine verkettete Liste von CPhysicalStone's an, verwaltet diese. Das Intro wird zu Beginn von der GUI erstellt, dieses wird bevorzugt von der GUI bearbeitet (wird über allem gerendert, erhält alle Mausklicks). Bei Mausklick, oder Tastendruck wird das Intro von der GUI entfernt.

TOptions (options.cpp)

Dient nur als Datentyp für Optionen, die sich für die GUI einstellen lassen. Wird nur einmal von der GUI instantiiert, und kapselt Zugriffe für andere Klassen.

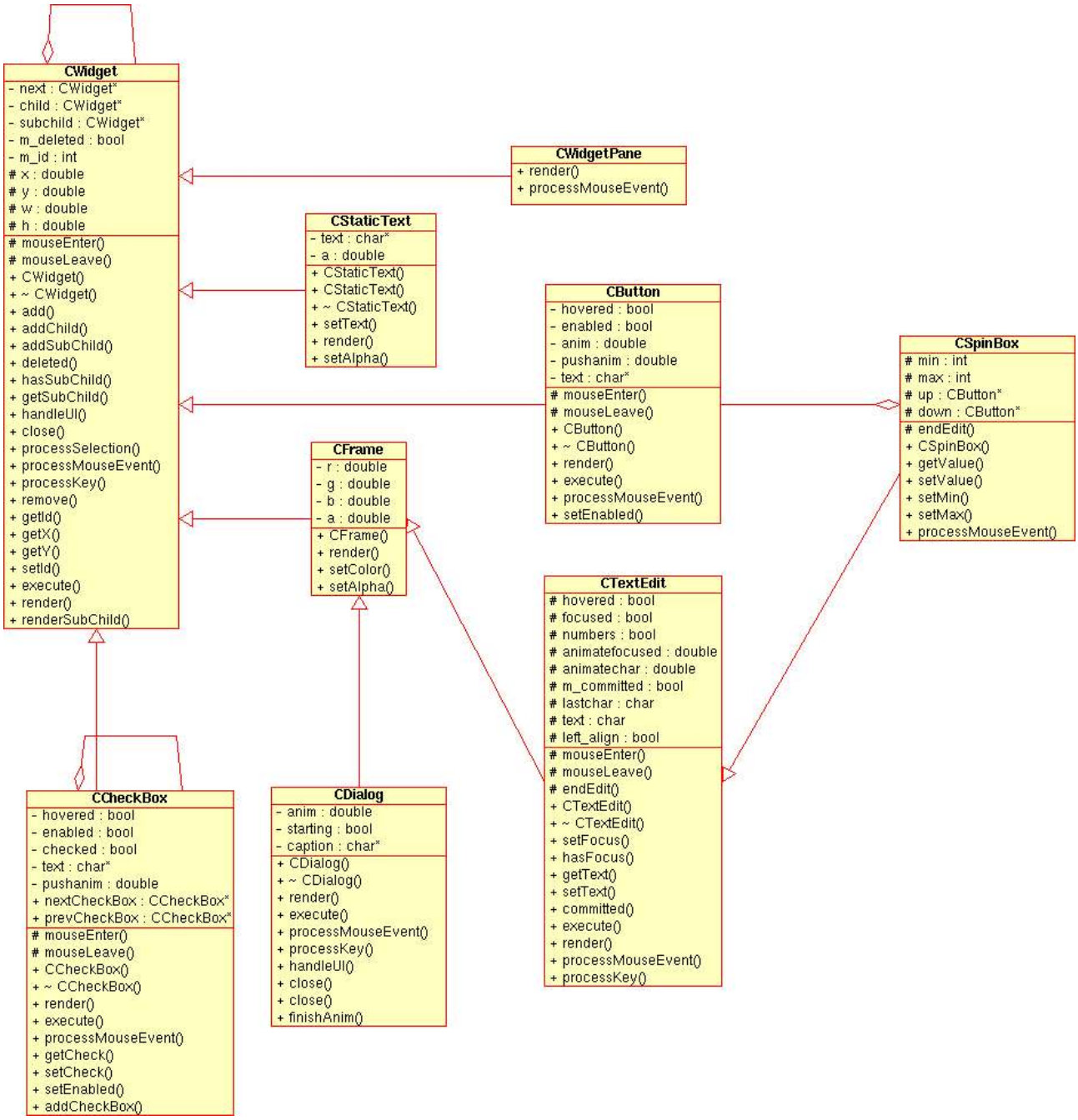
CGUI (gui.cpp)

Herzstück ist die Klasse CGUI (gui.cpp). Sie vereint alle Teile des Spiels:

- Sie erstellt zunächst ein CWindow.
- Sie beinhaltet eine CSpielClient Instanz für die Verbindung zum CSpielServer.
- Sie bietet die Möglichkeit, einen CSpielServer in einem sekundären Thread zu starten, also ein Spiel zu „hosten“, gleichermaßen für Netzwerkspiel, und lokales Singleplayerspiel.
- Bietet allen beteiligten Klassen Zugriff auf Optionen und Parameter des Spiels.
- Die Methode „run“ beinhaltet die hauptsächliche Spielschleife. Periodisch
  - Lässt sie den CSpielClient anstehende Netzwerknachrichten verarbeiten.
  - Lässt sie das CWindow anstehende Fensternachrichten verarbeiten. Events werden von der GUI verarbeitet.
  - In bestimmten Zeitabständen (angestrebt 35Hz.) rendert sie den aktuellen Spielstand und animiert alle Objekte (Effekte, Menüs, Intro, etc.). Dazu misst sie die Zeit, die für eine Periode der run() Schleife benötigt wird, und benutzt diese Zeit als Animationsfaktor.

CGUI beinhaltet alle Funktionen zum Rendern des Spielfelds und von Steinen, bereitet das Rendern vor, bearbeitet Mausereignisse, die die Spielsteine und das Feld betreffen.

7.Widgets



Die Menü-Engine ist hierarchisch aus verketteten Listen von Objekten des Typs CWidgets aufgebaut. CWidget ist eine abstrakte Klasse ohne besondere Funktionalität und repräsentiert ein 2D-Objekt des Menüs.

CWidget (widgets.cpp)

Jedes **CWidget** besitzt:

- Eine Position (x,y,w,h) im 2D Raum. Koordinaten werden immer als 0,0-640,480 behandelt, also 640 ist der rechte Rand des Fensters.
- Drei verkettete Listen:
    - Next: Zeiger auf das nächste Widget in derselben Ebene.
    - Child: Zeiger auf ein Unter-Widget, das sich innerhalb des Widgets befinden soll (z.B. ein CButton in einem CDialog ist ein child).
    - SubChild: Ein Unter-Widget, das sich aber eine Ebene über dem Widget befindet und möglichst modal ist (z.B. ein Unterdialog, der die Bedienung dieses Dialogs verhindern soll)
- Eine ID. Dies ist irgendeine Zahl, die es ermöglichen soll, das Widget bei Mauseingabe zu identifizieren. Ist nur nötig, wenn das Widget auf Mausereignisse reagieren soll. Beim Rendern in den Stencil-Buffer wird mit dieser ID als Name ein Quad an Position (x/y/w/h) gerendert, wodurch das Widget identifizierbar ist.
- Die Memberfunktion processMouseEvent(…). Sie wird bei jedem Mausereignis aufgerufen. Rekursiv wird das Ereignis zu allen untergeordneten Widgets durchgereicht und bei Behandlung wird die Kette abgebrochen. Rückgabewert ist eine Zahl (Kommando), worauf der Aufrufer reagieren kann.
- Die Funktion processKey verarbeitet einen Tastendruck und bricht ab, sobald die Taste verarbeitet wurde.
- In der Methode render() wird das Widget, mit allen next und child Widgets gerendert.
- In der Methode renderSubChild() wird nur das SubChild (und deren SubChilds) gerendert, damit die Reihenfolge in der Z-Ebene stimmt.
- Die restlichen Funktionen dienen der Verwaltung des CWidgets.

CWidgetPane

Die CWidgetPane existiert genau einmal in der Klasse GUI, leitet sich von CWidget ab, und dient als Kopf aller verketteter Listen. Alle Widgets werden von der CGUI der WidgetPane angehängt.
Beim Rendern bereitet sie den OpenGL Kontext für das 2D Rendern vor und verwaltet die Mausbehandlung.

CStaticText

Dieses CWidget rendert lediglich einen statischen 2D-Text an die übergebene Position, wahlweise zentriert oder transparent.

CFrame

CFrame ist ein Widget, das lediglich ein transparentes, farbiges Rechteck rendert, sonst nichts. Dient als Untergrund für Dialoge, Widgets, oder das Menü.

CDialog

Ein CDialog erweitert ein CFrame um eine Beschriftung und eine Animation beim Ein- und Ausblenden. Zudem schließt er sich automatisch beim Drücken von Escape und beim Kommando 1, also Betätigen eines CButtons mit der ID 1.

CButton

Das Widget repräsentiert einen Knopf im Menü. Er benötigt Koordinaten, den Text, und die ID, über die der Knopf erreichbar sein soll. Wird der Knopf gedrückt, gibt dessen processMouseEvent die ID als Kommando zurück, und der aufrufende Dialog kann den Kopf identifizieren. Der Knopf kann mit setEnabled ausgegraut werden.

CCheckBox

Eine Checkbox, die an oder aus sein kann. Erwartet, eine Position, den Text und eine ID. Die Position kann relativ zu einem anderen übergebenen Widget angegeben sein.
Mit getCheck und setCheck kann der Status gelesen/gesetzt werden. Die CheckBox gibt als Kommando ihre ID zurück, wenn sich der Status durch Klicken geändert hat.
Mit addCheckBox kann man der CheckBox weitere CheckBoxes anhängen, die in einer verketteten Liste verwaltet werden. Von allen mit einander verknüpften CheckBoxes kann immer nur eine aktiv sein (sog. RadioGroup).
Eine CheckBox kann mit setEnable ausgegraut werden.

CTextEdit

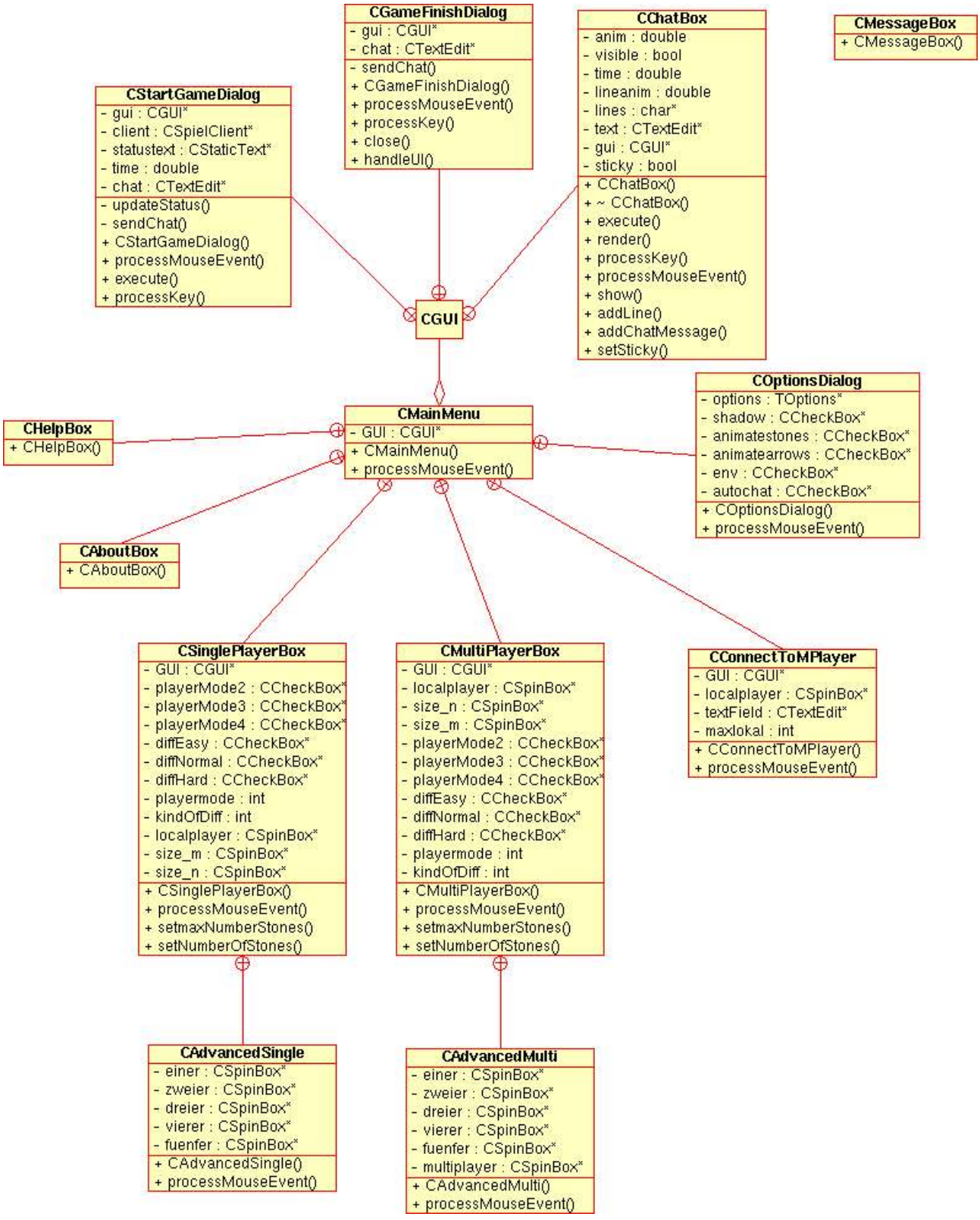
Ein Texteingabefeld, um den Benutzer Text an das Spiel übergeben lassen zu können. Es benötigt eine Position, eine ID, einen voreingestellten Wert und kann wahlweise nur Nummern akzeptieren, oder auch Text.
Der Text ist stets auf 80 Zeichen begrenzt, kann die ersten 127 Zeichen, sowie die deutschen Umlaute enthalten.
Mit setFocus kann man dem TextEdit direkt den Fokus geben, z.B. direkt beim Erstellen eines Dialogs.
Mit setText und getText erhält man Zugriff auf den Text des Felds.
Drückt man innerhalb eines TextEdit-Feldes Return, liefert ein Aufruf von committed() true zurück. Darauf reagiert z.B. die Chat-Box, um beim Druck auf Return den Text direkt zu versenden. Escape deaktiviert den Fokus.
Das processMouseEvent gibt NICHT IMMER die ID zurück, wenn sich der Inhalt des Edit-Feldes geändert hat. NICHT DARAUF VERLASSEN.

CSpinBox

Erweitert das CTextEdit und fügt diesem zwei CButton Objekte als Child hinzu. Über diese kann der Benutzer den Zahlenwert des TextEdits um eins erhöhen, bzw. erniedrigen. Das Feld akzeptiert nur Zahlenwerte, die über getValue und setValue erreichbar sind. Über setMin und setMax kann man die Grenzen des Wertes setzen.

8.Menü

Das Menü ist in folgende Teilbereiche aufgeteilt:



Alle diese Klassen, bis auf die CGUI leiteten sich von der Klasse CDialog ab. Somit haben alle die selben Eigenschaften und Animationen. Die häufigsten Objekte, die in den Dialogen verwendet werden sind CSpinBox, CCheckBox und CTextEdit. Das Menü stellt hauptsächlich die Schnittstelle zwischen dem Anwender und der grafischen Oberfläche dar. Mit Hilfe der oben erwähnten Objekten kann der Benutzer auf eine angenehme Art und Weise die verschiedenen Funktionen nutzen.

9.Die Spielinformationen

Um die Spielinformationen intern fest zu halten und zu verwalten dienen im Wesentlichen 3 Klassen:

1. Die Klasse CSpiel: Sie ist das Kernstück der Spielinformationen. In ihr liegt das aktuelle Spielfeld in Form eines Arrays vor und sie bietet Zugriff auf die teilnehmenden Spieler. Eine ihrer Hauptaufgaben ist es Züge auf ihre Gültigkeit zu überprüfen, und auf das Spielfeld zu übertragen, oder auch wieder zu entfernen.

2. Die Klasse CPlayer: Sie Speichert die Spielerinformationen, seine Spielsteine, seine möglichen Züge und bietet verschiedene Punktzählungen.

3. Die Klasse CStone: Hier ist die Form eines jeden möglichen Steins gespeichert. Sie dreht und spiegelt Spielsteine und merkt sich auch wie oft ein Stein noch vorhanden ist.

Das Spielfeld (CSpiel, spiel.cpp)

Die Spielfeld wird in 5 verschiedenen char-Arrays gespeichert. Bei einem 20x20 Feld sind es also 5x20x20 chars die benötigt werden um eine Spielsituation zu Speichern.
Aber wozu 5 Felder? Warum nicht einfach eins?!

Das erste Feld:
Speichert die allgemeinen Daten, die zum Anzeigen des Spielfelds benötigt werden. Also, die Farbe des Steins, der sich auf einem bestimmten Feld befindet, bzw ob das Feld frei ist.

Das zweite bis fünfte Feld:
Speichert jeweils die speziellen Daten für den ersten bis vierten Spieler. Hier liegen Informationen, über unerlaubte Felder und mögliche Anlegepunkte des jeweiligen Spielers. Durch diese 4 zusätzlichen Felder können wir uns also Mehrfachberechnungen dieser Informationen sparen.

Der Vorteil dieser Aufteilung in 5 Felder ist, dass eine Zugüberprüfung sehr schnell geht. Es Muss lediglich ein Stein an eine beliebige stelle gelegt werden. Wenn mindestens ein Steinelement auf einem Feld platziert wird, dass für diesen Spieler als unerlaubt gilt, oder kein Steinelement auf einem Feld platziert wurde, an dem er anlegen darf, so ist der gesamte Zug ungültig.
Die Abfragen beziehen sich durch die 5 Felder nur auf simple Arrayzugriffe. Keine weiteren Schleifen und keine aufwändigen Berechnungen.

Nachteil des Ganzen war allerdings ein fünffacher Speicherverbrauch gegenüber der Variante die nur das erste Array verwendet und alle anderen Informationen spontan

errechnet.

Umgehen des Nachteils:

Wir haben daher etwas rumgetüftelt und eine Möglichkeit gefunden unsere Informationen die wir in 5 char-Arrays speichern in einem einzigen char-Array zu speichern. Um dies zu Bewerkstelligen nutzen wir jedes einzelne Bit eines jeden chars:
Ein char besteht aus 8 Bit, somit bleiben für jeden der vier Spieler genau 2 Bit, mehr als genug.

Das erste Bit eines Spielers gibt an ob es ein verbotenes Feld ist. Ist dieses Bit auf 1 gesetzt, so darf der Spieler keinen Stein mehr dort platzieren, entweder weil der Platz bereits belegt ist, oder weil ein eigener Stein das Feld mit einer Fläche berührt.
Das zweite Bit gibt an ob es sich um einen erlaubten Anlegepunkt handelt. Ist es gesetzt, so kann der Spieler dort einen Stein anlegen.

Stehen beide Bits auf 0 so handelt es sich für diesen Spieler um ein freies Feld, ohne Sonderregelungen.
Dass beide Bits auf 1 stehen ist ein unmöglicher Zustand, da kein unerlaubtes Feld ein mögliches Anlegefeld sein kann.

Array 2 bis 5 wären somit untergebracht, fehlt noch das erste, welches lediglich die Farben, der dort liegenden Steine, beinhaltet. Da keine Bits mehr frei sind bedienen wir uns dem, gerade beschriebenen, unmöglichen Zustand der doppelten 1.

Bei einem Belegtem Feld legen wir die ersten 6 Bit als 1 fest, somit ist für alle Spieler klar, dass es ein verbotenes Feld ist. Die letzten beiden Bit geben nun die Spielernummer zurück, der dort einen Stein liegen hat.

00 bei Blau,
01 bei Rot,
10 bei Gelb,
11 bei Grün.

Auf diesem Wege können wir 5 Arrays benutzen um bessere Performance zu gewährleisten, ohne dabei mehr Speicherplatz zu verbrauchen.

Ein paar Beispiele zur Verdeutlichung:

10 01 00 00b => Grün darf in diesem Feld nicht legen. (10b)

Gelb kann sogar in diesem Feld anlegen (01b)
Für Rot und Blau ist dies ein normales, freies Feld (00b)

11 11 11 01b => Auf diesem Feld liegt ein Stein (11 11 11b). Es ist also für alle Spieler nicht erlaubt dort einen Stein abzulegen.
Rot (01b) hat hier einen Stein liegen

Züge auf Gültigkeit überprüfen

Die Aufteilung in 5 Arrays macht die Zugüberprüfung zu einem Kinderspiel. Man legt lediglich die Form des Steins virtuell über das Spielfeld und überprüft jedes Feld auf dem er liegen würde.
Dazu muss man die entsprechenden 2 Bits überprüfen die zu dem jeweiligen Spieler gehören. Ein Feld ist ungültig, wenn es belegt ist (ersten 6 Bit stehen auf 1), oder wenn die Bits des Spielers 10b anzeigen. Ist mindestens ein Feld auf diese weise ungültig, so ist der gesamte Zug ungültig.
Außerdem muss mindestens ein Feld den Bitstatus 01b (Anlegepunkt) erfüllen. Ist das der Fall so ist der Zug gültig.

Steine ablegen

Das Ablegen der Steine ist eine einfache Sache, und eigentlich nicht der rede Wert. Wenn ein Zug durch die Gültigkeitsüberprüfung durch kommt, kann der Stein einfach abgelegt werden.
Dazu wird das Spielfeld-Array an den entsprechenden Stellen mit 111111xxb aktualisiert. Um jetzt noch die speziellen Spielerinformationen zu berücksichtigen müssen bei dem Ablegen der einzelnen Steinelemente die Nachbarfelder ebenfalls verändert werden. Durch Bitoperationen werden die Bits des Spielers bei einer Ecke mit 01 (anlegen möglich) und eine Fläche mit 10 (unerlaubtes Feld) gekennzeichnet werden.

Spielerinformationen (CPlayer, player.cpp)

CPlayer steht stellvertretend für einen Spieler. Hier liegt ein Satz von Spielsteinen (siehe CStone) vor und es können einige Statistiken zu diesem Spieler aufgerufen werden, welche die Ki benutzen kann um Punktwertungen aufzustellen.
Außerdem merkt sich CPlayer die Spielernummer, und die Spielernummer seines Verbündeten, sofern er einen hat.

Diese Klasse bietet also eigentlich nicht viel mehr als einen Datenspeicher und einige Zählmethoden. Hier wird sie nur erwähnt, da sie ein Array von CStones beinhaltet, in dem 21 CStones gespeichert werden, nicht mehr und nicht weniger.
Doch was genau ist ein CStone?!

Spielsteine (CStone, stone.cpp)

CStone realisiert die interne Darstellung der Spielsteine. Dabei ist es ganz interessant, dass dafür lediglich 4 Variablen benötigt werden. Die Steine brauchen *nicht* jeweils ein eigenes Array, in denen ihre Informationen gespeichert werden.

Eine Variable für:

1. Die Verfügbarkeit (Available):
Hier wird gespeichert wie oft der Stein noch existiert. Wird ein Stein abgelegt muss Available um eins reduziert werden. Ist Available = 0 so steht der Stein nicht mehr zur Verfügung.

2. Die Form des Steins (Shape):
Jeder Steinform wurde eine Nummer von 0 bis 20 zugeteilt, die stellvertretend für die Form des Steins stehen soll. Sie ist also eine Art Zugriffsschlüssel.
Die Form des Steins wird also einfach als Nummer von 0 bis 20 gespeichert. Bei jedem CPlayer sollte es so sein, dass der Index des CStone immer auch seinem Wert in Shape entspricht, dies ist allerdings nicht Voraussetzung für die Funktionsfähigkeit des Programms.

3. Die Anzahl der Rotationen (Rotatecounter):
Jeder Spielstein verfügt über einen Rotatecounter, der einfach nur mit zählt wie oft der Stein gedreht wurde. Eine Rotation im Uhrzeigersinn erhöht den Wert um eins, eine Rotation gegen den Sinn verringert den Wert um eins.
Um das ganze einfacher zu halten wiederholen sich die werte, so dass sie immer in einem Positiven Bereich von 0 bis 3 liegen, bei einigen Steinen sogar nur von 0 bis 1 oder lediglich 0.
Auf diese Art ist jeder möglichen Drehung des Steins ein eindeutiger Wert zugeordnet was eine Rekonstruktion des Steinzustands sehr einfach macht. Außerdem wird durch dieses System eine Drehung des Steins sehr einfach. Es wird nämlich intern gar nichts gedreht, es wird lediglich der Rotatecounter um eins erhöht oder verringert.

4.Die Anzahl der Spiegelungen (Mirrorcounter) :
Der Mirrorcounter zählt die Anzahl der Spiegelungen mit und funktioniert im großen und ganzen genau wie der Rotatecounter. Hier allerdings ist der Minimalwert 0 und der Maximalwert (für die meisten Steine) 1.
Wird horizontal gespiegelt muss der Wert nur invertiert werden. Wird vertikal gespiegelt muss der Rotatecounter zusätzlich um 2 erhöht werden.
Und schon hat man eine Spiegelung des Steins ohne Arrayoperationen durchführen zu müssen.

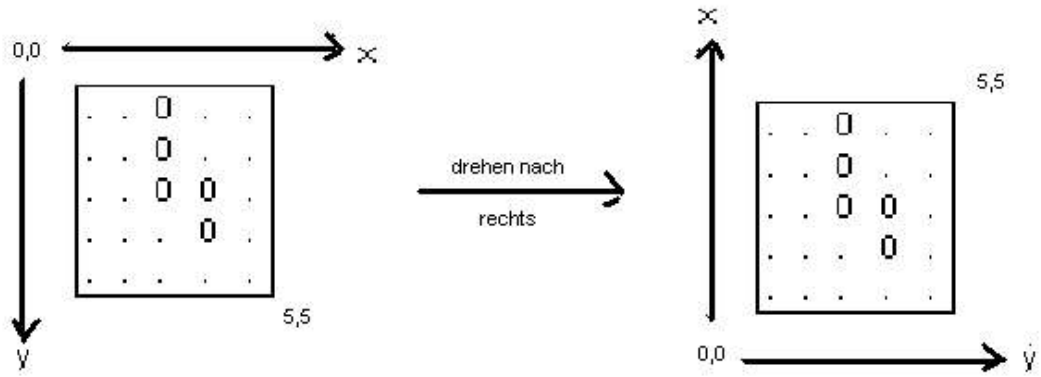
Stellt sich die Frage wann der Stein denn nun tatsächlich gedreht wird, und wo seine Form Definiert wird.

Die Steinkonstanten

Für jede der 21 Formen gibt es ein 5x5 Array an Konstanten. So ergibt sich ein 21x5x5 Array das die Formen von allen 21 Steinen beschreibt. Dieses Array existiert im Speicher nur ein mal, und steht der Klasse CStone zur Verfügung.
Über die oben erläuterte Variable Shape und unter Zuhilfenahme von x und y Koordinaten kann jeder CStone auf seine tatsächliche Form zurückgreifen.
X,Y = 0,0 wird dabei als obere, linke Ecke Interpretiert; x,y = 5,5 wird als untere, rechte Ecke interpretiert.

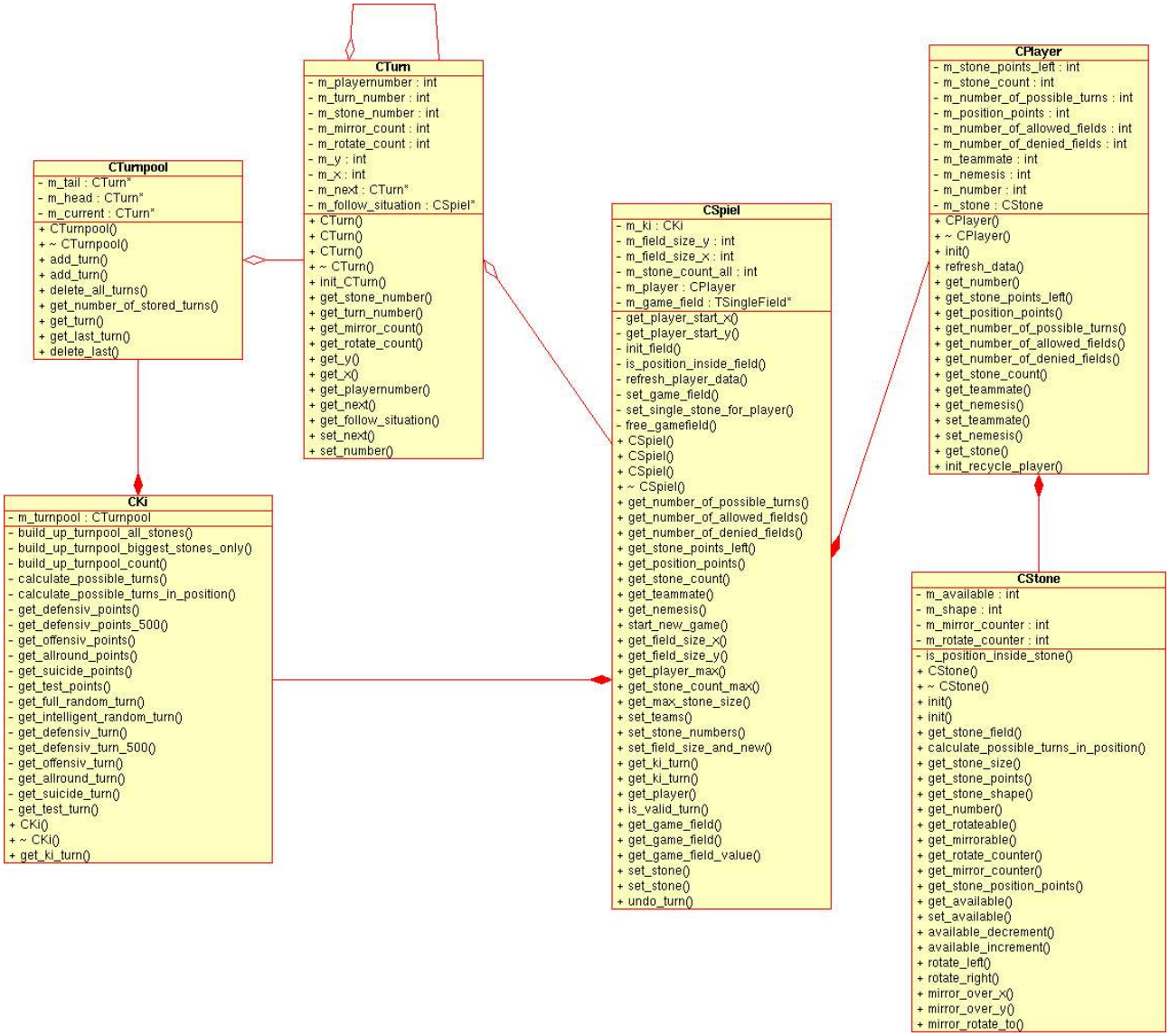
Sollte ein CStone rotiert worden sein (Rotatecounter ist größer als 0), so dreht sich dadurch der Stein nicht wirklich, sondern es ändert sich nur die Interpretation der Positionskoordinaten.
So könnte nun x,y = 0,0 die untere, rechte Ecke beschreiben und x,y = 5,5 die obere, linke Ecke.

Um einen Stein zu drehen, wird also nicht der Inhalt der Steinarray gedreht, sondern es wird die Achsenbezeichnung in entgegengesetzte Richtung gedreht.



Die Spiegelung funktioniert genauso, nur dass hier die Achsenbezeichnungen gespiegelt werden.

10.Der Computergegner (CKi, ki.cpp)



Bei der Entwicklung des Computergegners haben wir besonders darauf geachtet, dass er mit möglichst allgemeinen Methoden arbeitet, die in jedem der von uns implementierten Spielmodi funktionieren.

Die Klasse CKi zeigt nach außen hin dabei nur eine einzige Funktion, get_ki_turn(), die einen Spielzug vom Typ CTurn zurückgibt, der dann von anderen Klassen verwendet werden kann.

Der lange Weg zum Computerzug:

1. Aufbau des Turnpools

Zunächst muss der Computer wissen welche Züge er überhaupt zur Verfügung hat. Zu diesem Zweck wird beim Aufruf von get_ki_turn() als erstes das gesamte Spielfeld nach möglichen Anlegepunkten für den jeweiligen Spieler überprüft. Ist einer dieser Punkte gefunden, so wird jeder noch verfügbare Stein dieses Spielers, in jeder möglichen Dreh- und Spiegelkombination dort angelegt. Ergibt sich daraus ein gültiger Zug, wird dieser in einer Liste abgespeichert die wir CTurnpool genannt haben.
Theoretisch ergibt sich aus dieser Vorgehensweise ein Turnpool, in dem sich alle erlaubten Züge befinden. Da allerdings der Aufbau dieses Turnpools und besonders die spätere Weiterverarbeitung viel Rechenzeit verbrauchen wird, wird eine Vorauswahl der Züge getroffen. Um dies zu tun, werden lediglich die Züge der 10 größten Steine, mit Anlegemöglichkeiten, im Turnpool zwischengespeichert.
Diese Einschränkung liefert noch immer sehr gute Züge und bewirkt gerade am Anfang des Spiels, wo die Rechenzeiten noch lang sind, eine deutliche Beschleunigung.

2. Die Folgesituationen der möglichen Züge

Aus jedem, im Turnpool abgelegtem, Zug und der aktuellen Spielsituation ergibt sich eine Folge-Spielsituation. Diese Folgesituation wird im nächsten Schritt errechnet, wobei eine Kopie der Spielfeldinformation erstellt werden muss, und zwar für jeden Zug der sich im Turnpool befindet. So kann es schon mal passieren dass diese Prozessoraufwendige Operation über 2000 mal durchgeführt wird, während der Computergegner ’überlegt’. Hier liegt also zweifellos die Phase die in der Berechnung am meisten Zeit in Anspruch nimmt.

3. Die Auswertung der Folgesituation

Während die Folgesituation errechnet wird, werden auch sofort einige Punktzahlen für jeden Spieler berechnet. So wird zum Beispiel festgehalten wie viele mögliche Züge, oder wie viele Anlegepunkte ein Spieler noch hat.
Diese Werte können nun von der Klasse CKi beliebig kombiniert werden um damit eine Punktwertung für jeder Folgesituation zu erstellen. Der Zug mit der höchsten Punktzahl wird dann als bester Zug angenommen.
Es ist jedoch unwahrscheinlich das dieser Zug tatsächlich der beste Zug ist, denn Blokus ist ein sehr komplexes Spiel. Die 100%tig richtigen Bewertungskriterien zu finden ist unmöglich. Ich glaube jedoch, dass sich mit dieser Methode zumindest ein sehr guter Zug errechnen lässt.

4. Computergegner-Schwierigkeit und Zufall

Um das ganze interessanter für den Spieler oder Zuschauer zu machen rechnet die Klasse CKi bei der Punktauswertung der Züge zufällig bis zu fünf Prozent des Jeweiligen Punktwertes drauf. Dadurch ist es möglich, dass, wenn es mehrere Züge mit guter Punktwertung gibt, ein zufälliger von diesen ausgewählt wird. Jedoch bleibt die Wahrscheinlichkeit, dass es der ‚beste’ Zug wird immer noch am höchsten.

Für die leichteren Schwierigkeitsgrade lässt sich einfach der benutze Prozentsatz erhöhen. Der mittlere Computergegner benutzt beispielsweise eine Abweichung von bis zu 20 Prozent und der leichte sogar 75 Prozent. Da immer vom jeweiligen Punktwert ausgegangen wird, werden auch die leichteren Computergegner gute Züge machen, jedoch machen sie auch wesentlich mehr 'Fehler', dadurch dass die Zufallszahlen größer werden.

Bewertungskriterien der Computergegner

Folgende Bewertungskriterien beachtet unser Computergegner zur Zeit. Sie werden teilweise noch mit verschiedenen Gewichtungen multipliziert, um ihre Wichtigkeit hervorzuheben, auf diese Gewichtungen wird im folgenden allerdings nicht eingegangen.

Zur Punktebewertung gehören

1. Die Anzahl, der eigenen möglichen Züge.
Der Computergegner versucht allein dadurch schon selbstständig vom Spielfeldrand weg zu kommen und baut somit in die Mitte. Gleichzeitig, und darum geht es eigentlich bei diesem Kriterium, versucht er sich selbst möglichst wenig zu verbauen und möglichst viele neue Anlegemöglichkeiten zu erschließen.
Außerdem wird durch dieses Kriterium bewirkt, dass Steine mit Wenigen verbleibenden Anlegemöglichkeiten bevorzugt gesetzt werden.

2. Die eigenen Anlegemöglichkeiten werden Modifiziert durch die Form des Steins. Je mehr Möglichkeiten es gibt den jeweiligen Stein zum Spiegeln oder zu Drehen, desto kleiner ist dieser Modifikator.
Hierdurch wird verhindert, dass die Ki Steine bevorzugt verbaut, die sich weder drehen noch spiegeln lassen. In der Folgesituation würden dies Steine nämlich den geringsten Punkteverlust bedeuten, da sie die wenigsten Anlegemöglichkeiten hätten.

3. Die Anlegemöglichkeiten werden gewichtet mit der Größe des jeweiligen Steins.
Hat ein großer Stein viele Möglichkeiten, wird dies dadurch besser Bewertet als wenn ein kleiner Stein viele Möglichkeiten hat.

4. Die Größe des Steins aller verbleibender Steine (negativ).
Dies muss mitbeachtet werden, da sonst kleine Steine zu früh gelegt werden würden.

5. Die ersten drei Punkte (Anlegemöglichkeiten, modifiziert durch Form des Steins und Größe des Steins) werden auch für jeden Gegner angewendet, natürlich als Negativwert. Die Ki verbaut dem Gegner seine besten Möglichkeiten, wobei ein verbauter großer Stein wieder mehr zählt als ein verbauter kleiner Stein. Dadurch wird der Gegner in die Enge getrieben.

6. Anlegepunkte des Gegners (negativ).
Hierdurch werden Zugmöglichkeiten des Gegenspielers von der Ki im Keim erstickt. So kann es schneller passieren, dass sie „einen Riegel" vor den Expansionsversuchen des

Gegners vorschiebt.

7. Entfernung zum Startpunkt.
Ein etwas umstrittener Bewertungspunkt. Doch wir glauben rausgefunden zu haben, dass es wichtig ist möglichst weit in andere Gebiete vorzudringen, egal was sich dabei in der eigenen Startzone tut. Die Theorie ist die, dass man sich beim legen auf engem Raum viele Möglichkeiten selbst verbaut. Baut man weit weg, so verschlechtert sich wenigstens die Situation nicht durch die eigenen Züge.
Wie auch immer, wenigstens baut der Computergegner durch dieses Kriterium auch in die Mitte.

8. Verbündete Spieler werden gar nicht berücksichtigt.
Ihnen wird nicht Aktiv etwas verbaut, doch es wird auch nicht darauf geachtet, dass man ihnen nichts Verbaut. Würde man es anders machen, so würden sich daraus Gassen ergeben, da beide Verbündeten aufpassen, dass sie sich nicht zu nahe kommen. Diese wären dann bevorzugte Angriffspunkte des Gegenspielers, da dort ungehindertes Durchkommen ist und man beiden Gegnern gleichzeitig Möglichkeiten verbauen kann.

11.Die neuen Anforderungen

Im Laufe des Projektes wurden neue Anforderungen an die Software gestellt:

- Eine variable Spielfeldgröße
- Damit verbunden eine Einstellung der Spielsteinzahl
- Und eine Zugzurücknahme

Die Änderungen waren verhältnismäßig leicht umzusetzen.

Die variable Spielfeldgröße

Da wir die Spielfeldgröße schon vorher als Konstante mitgeführt hatten brauchten wir lediglich die Konstante zu einer Variablen machen.
Auch für die Ki mussten wir keine Änderungen vornehmen, da diese sich zu keiner Zeit auf eine bestimmte Spielfeldgröße konzentriert hat.

Die Einstellung der Spielsteinzahl

Auch hier waren nur kleine Änderungen erforderlich. Der Datentyp des Available-Wertes eines CStones, der anzeigt ob ein Stein noch verfügbar ist, oder schon gelegt wurde, musste ledig von boolean auf integer geändert werden.
Nun speichert der Available-Wert die Anzahl der noch verfügbaren Steine dieses Typs.

Die Zugzurückname

Hier wurde es etwas komplizierter, da es durch mehrere Ebenen des Programms ging. Es wurde in CSpiel eine Rücknahmefunktion hinzugefügt und im CSpielleiter musste eine Liste aller gesetzten Züge hinzugefügt werden. Für diese Liste konnten wir jedoch glücklicherweise den CTurnpool benutzen den wir ohnehin schon für die Ki benötigt hatten.

Alles im allem waren die Änderungen für unsere Gruppe also kein großes Problem.