Grundlagen der Informatik III

Wintersemester 2015/2016

Oskar von Stryk, Alexander Stumpf, Max von Buelow

TECHNISCHE UNIVERSITÄT DARMSTADT

4. Praktikum

Ausgabe: 20.01.2016; Abgabe: 07.02.2016 23:55 Uhr

Abgabe der Programme: Ihre Abgabe soll sämtliche zum Kompilieren und Ausführen Ihres Programms benötigten Dateien in einer .zip- oder .tar.gz-Datei komprimiert enthalten und im Moodle-Kurs unter der entsprechenden Abgabe hochgeladen werden.

Hinweis: Alle folgenden Aufgaben beziehen sich auf den **C++-Standard (bis C++11)**. Beachten Sie, dass für die Bearbeitung der Aufgaben keine externen Bibliotheken mit Ausnahme der C++-Standard-Bibliotheken, Lex und Yacc bzw. Flex und Bison benutzt werden dürfen.
Ein Grundgerüst zu diesem Praktikum können Sie im Moodle-Kurs herunterladen. Als Abhängigkeit sind Bison und Flex vorausgesetzt, die jeweils auf den RBG-Rechnern vorinstalliert sind. Die Bearbeitung der Aufgaben sollte auch mit eigenen Rechnern problemlos möglich sein, allerdings geben wir hierfür keine Gewährleistung.

Einführung

In diesem Praktikum wollen wir einen Compiler für eine virtuelle Voxel-Rendering-GPU implementieren. Der Compiler muss in der Lage sein, eine fiktive Voxelsprache zu interpretieren und für die GPU einen korrekten Maschinencode zu erzeugen. Dazu müssen die in der Vorlesung vorgestellten Phasen eines Compilers umgesetzt werden:

1. Lexikalische Analyse
2. Syntaktische Analyse
3. Semantische Analyse
4. Zwischencodegenerierung
5. Codeoptimierung
6. Erzeugung von Maschinencode

Ihr Compiler soll eine Eingabedatei aus rein boolesche Ausdrücken einlesen, welche in einem Voxelgitter ausgewertet und anschließend in Dreiecke tesseliert werden, sodass man am Ende eine 3D-Datei im *Polygon File Format*(*.ply) erhält.

Da die genannte GPU nicht real existiert, wird ein Simulator benötigt, um ihre Implementierung zu testen. Hierzu wird der Simulator *sim* mitgeliefert, welcher den Maschinencode interpretieren kann und daraus ein ply-File generiert, das sie z.B. mit den Programmen Meshlab oder Blender bzw. mit dem 3D-Builder unter Windows 10 ansehen können.

Wie auch in Übungsblatt 11 werden hier die Tools *Flex* und *Bison* zum Einsatz kommen, die kompatibel zu *Lex* und *Yacc* aus der Vorlesung sind.

Syntax

Die Syntax hat C-ähnliche Struktur bestehend aus Funktionen und Unär- sowie Binäroperatoren. Funktionsnamen müssen sich aus dem regulären Ausdruck [a-zA-Z0-9]+ ableiten lassen. Die Funktionssignatur ist eingegrenzt durch Klammern, in denen die Argumente $a_i \in \mathbb{N}_0$ durch Kommata getrennt aufgezählt werden. Die Funktionsargumente a_i können in Dezimal- sowie Hexadezimaldarstellung (gekennzeichnet mit 0x als Präfix) dargestellt werden. Eine Funktion definiert hierbei eine parametrisierte Menge von Voxeln $V_i \subseteq \mathbb{N}_0^3$. Es sollen in der Grammatik weder die Funktionsnamen noch die

Anzahl der Argumente eingeschränkt werden.

In unserer Voxelsprache sind folgende Operatoren über Voxelmengen definiert:

Operator	Definition	Beschreibung
| (binär)	$V_{result} = V_1 \cup V_2$	Disjunktion: Vereinigung zweier Voxelmengen
& (binär)	$V_{result} = V_1 \cap V_2$	Konjunktion: Schnitt zweier Voxelmengen
! (unär)	$V_{result} = \mathbb{N}_0^3 \backslash V_1$	Komplement: Differenz von $\mathbb{N}_0^3$ zu der Eingabemenge

wobei $V_1, V_2, V_{all}, V_{result} \subseteq \mathbb{N}_0^3$ gilt. Hierbei sind V_1 und V_2 die Eingabemengen und V_{result} die Ergebnismenge. Um Mehrdeutigkeiten manuell zu vermeiden, wird analog zu arithmetischen Ausdrücken die Infixnotation eingesetzt.

Leerzeichen (*spaces*, *tabs*, *carriage returns* und *newlines*) sollen ignoriert und die Semantik nicht beeinflussen. Jede Erweiterung der Syntax ist gerne gesehen, so lange sie nicht destruktiv ist. Zum Bestehen muss jedoch immer gewährleistet werden, dass ihr Compiler die oben genannte Spezifikation korrekt interpretieren kann und jede unserer Testeingaben zum korrekten Ergebnis führt!

Aus praktischen Gründen wird in unserer Implementierung keine konkrete Voxelmenge berechnet. Stattdessen durchlaufen wir das gesamte Voxelgitter mit einer Schleife und prüfen für jeden Voxel, ob er innerhalb der zu zeichnenden Funktion liegt.

Eine Beispielcode der obigen Syntax sieht wie folgt aus:

```
1 (sphere(50 50 40 15 0xff0000) & !(box(35 35 40 10 10 10 0x0))) | heart(20, 20, 20,
    10, 0xdd0000)
```

Kompilieren

Ihren Compiler bauen Sie mit den folgenden Anweisungen. Sie benötigen in beiden Fällen ein installiertes CMake.

Linux mit gcc:

```
1 $ cd build
2 $ cmake ..
3 $ make
```

Alternativ können Sie sich auch das Projekt mit einer IDE ihrer Wahl (z.B. QtCreator) öffnen, bearbeiten und kompilieren. Entsprechende Anleitungen wie sie cmake-Projekte mit verschiedenen IDEs öffnen können, finden Sie im Internet.

Windows mit Visual Studio:
Für Windows befindet sich im Ordner „build_win“ ein Script namens „win_cmake.bat“. Nach dem Ausführen des Scripts können Sie anschließend das MS-Visual-Studio-Projekt „Project.sln“ öffnen. Unter `http://sourceforge.net/p/winflexbison/wiki/Visual%20Studio%20custom%20build%20rules/` finden Sie desweiteren eine Anleitung, wie man Lexer/Bison-Code besser debuggen kann.
Hinweis: Für ältere Visual Studio Versionen müssen Sie die „win_cmake.bat“ entsprechend anpassen. Visual Studio 2015 Community Edition bekommen Sie kostenlos direkt von der Microsoft Seite. Die Enterprise Edition kann von Microsoft DreamSpark (MSDNAA) kostenlos bezogen werden, sofern ihr Fachbereich am Dreamspark Programm teilnimmt.

Alle im folgenden verwendeten Dateien des Compiler Projekts befinden sich im Unterordner „compiler“. Die Vorlage ist soweit vorbereitet, dass Sie nur die angegebenen Stellen editieren müssen. Es ist nicht notwendig weitere Klassen oder Variablen anzulegen, um die Aufgaben zu lösen.

Aufgabe 1: Lexikalische Analyse

Fügen Sie erst in die „parser.y“ ihre benötigten Tokens ein. Vervollständigen Sie anschließend das Lex-Programm in der Datei „lexer.l“, um die oben definierten Lexeme zu erkennen. Beachten Sie hierbei folgende Hinweise:

1. Sie erhalten die durch ihren regulären Ausdruck erkannte Eingabe als `char*` in der Variable *yytext*. Achtung: *yytext* ist nicht persistent! Nach Aufruf des Lexers ist der Inhalt an der angegebenen Speicheradresse nicht mehr verfügbar.
2. Das **Attribut** eines Tokens kann in der Variable *yylval* hinterlegt werden. Die Elemente bzw. verwendeten Subtypen sind von der Union in „parser.h:19“ definiert.
3. Mit `atoi` konvertieren Sie eine als in `char*` dargestellte Zahl in einen `int`.
4. Mit `strtol` konvertieren Sie einen in `char*` dargestellte Hexadezimalzahl in einen `int`.
5. Mit `strdup` können Sie den Speicherinhalt an der Adresse `char*` duplizieren. Es wird hierfür automatisch ein neuer Speicherbereich allokiert.

Eine Zahl zwischen zwei und vier kann durch folgenden Ausdruck in ein `INT`-Token übersetzt werden, wobei dessen Attribut in *yylval* gesichert wird:

```
[2-4]         { yylval.ival = atoi(yytext); return INT; }
```

Zum Testen ihres Tokenizers wird standardmäßig der erkannte Tokenstream auf die Konsole ausgegeben. Sobald Sie die Aufgabe beendet haben, kommentieren Sie bitte unbedingt die folgende Zeile in der „parser.y“ ein:

```
//#define LEXER_IMPLEMENTED
```

Aufgabe 2: Syntaktische Analyse

In dieser Aufgabe wenden wir das Prinzip der *syntaxgerichteten Übersetzung* aus der Vorlesung an. Hierbei soll während dem Generationsprozess in C++ eine Datenstruktur aufgebaut werden, die unseren Syntaxbaum repräsentiert. Zu diesem Zweck müssen die Tokens in die entsprechende C++-Datenstrukturen und -Werte überführt werden. Hierzu wurden in der „parser.h“ die Klassen `Expr` für Knoten und `Fn` für Blätter angelegt. Die verwendeten Opcodes sind als Enum in der „ops.h“ vordefiniert und sollen hier verwendet werden. Das untere Beispiel zeigt die Produktionsregel für die Konjunktion.

```
expr:
    expr CONJ expr { $$ = new Expr(OP_CONJ, $1, $3); }
```

Zur Lösung der Aufgabe muss der Parser in der Datei „parser.y“ fertig implementiert werden, sodass er mithilfe einer Grammatik die oben erklärte Syntax erkennen kann und den Syntaxbaum erstellt. Definieren Sie dazu zunächst die Typen und anschließend die Grammatik selber. Die Implementierung ist hier für *Bison*, dessen Syntax aber kompatibel zu der in der Vorlesung vorgestellten *Yacc*-Syntax ist. Am Ende können Sie auch in diesem Fall ihre Implementierung anhand der automatischen Konsolenausgabe validieren.

Aufgabe 3: Semantische Analyse

Ihnen ist sicher aufgefallen, dass manche Funktionen eine unterschiedliche Anzahl von Argumenten besitzen und beliebige Namen haben können. In dieser Phase wird genau dies geprüft und eingeschränkt. Wenn die Semantische Analyse fehlschlägt, soll ein Fehler ausgegeben werden und der Compiler terminieren. Die Funktionen sind wie folgt definiert (Platzhalter für die Argumente sind hier in Großbuchstaben dargestellt):

1. $sphere(X, Y, Z, RADIUS, COLOR)$: Zeichnet eine Kugel an den Koordinaten X, Y, Z mit dem Radius $RADIUS$ und der Farbe $COLOR$.

2. $box(X, Y, Z, WIDTH, HEIGHT, DEPTH, COLOR)$: Zeichnet einen Kasten an den Koordinaten X, Y, Z mit den Dimensionen $WIDTH$, $HEIGHT$ und $DEPTH$ und der Farbe $COLOR$.
3. $heart(X, Y, Z, RADIUS, COLOR)$: Zeichnet ein Herz an den Koordinaten X, Y, Z mit dem Radius $RADIUS$ und der Farbe $COLOR$. Die Farbe ist in Hexadezimaldarstellung dargestellt.

Die Farben sind in Hexadezimaldarstellung in 8 Bit RGB-Kodierung dargestellt. So sind z.B. die Komplementärfarben als 0xff0000 (rot), 0x00ff00 (grün) und 0x0000ff (blau) kodiert.

In diesem Schritt müssen Sie überprüfen, ob die Syntax gültige Funktionsnamen hat und die Argumentlisten die richtige Längen haben. Hierzu müssen Sie in der „parser.h“ die Methoden `Fn::check(uint32_t &dim)` sowie die `Expr::check(uint32_t &dim)` implementieren, welche initial von `Ast::check(uint32_t &dim)` aufgerufen wird.
Hinweis: Wie bereits in der vorigen Aufgabe beschrieben, repräsentiert hierbei die Klasse `Expr` einen Knoten und `Fn` ein Blatt im Syntaxbaum. Wenn der Test fehlschlägt soll eine aussagekräftige Fehlermeldung auf der Konsole erscheinen und `false` zurückgegeben werden.
Hinweis: Der Syntaxchecker wird vom Compiler automatisch aufgerufen und erzeugt ebenfalls eine Konsolenausgabe. Analog zu den ersten beiden Aufgabe können Sie also problemlos ihre Implementierung testen.

Das zusätzliche Argument `uint32_t &dim` stellt eine **optionale** Herausforderung für Sie dar. Zusätzlich zur semantischen Prüfungen können Sie in der `check`-Methode die maximale Ausdehnung (Maximum entlang der X,Y und Z-Achse) des Voxelgitters bestimmen. Das Ergebnis eines Rekursionsaufrufs können Sie über die Variable `dim` zurückgeben. Falls Sie nicht in der Lage sind das Problem zu lösen, dann dürfen Sie die `Ast::check(uint32_t &dim)` in der „parser.h“ so umschreiben, dass diese den Konstanten Wert `128` für `uint32_t &dim` zurückliefert.

Aufgabe 4: Zwischencodegenerierung und Codeoptimierung

Codeoptimierung ist ein sehr komplexes Thema und verbleibt hier daher als optionale Aufgabe. Wir halten Ihnen die Möglichkeit offen, mithilfe einer geschickt geänderten Aneinanderreihung der Maschinenbefehle oder Optimierung des Syntaxbaums einen Geschwindigkeitsgewinn zu erzielen.
Die Zwischencodegenerierung wird in diesem Praktikum ebenfalls übersprungen, da wir aus den oben besagten Gründen keine Codeoptimierung durchführen wollen und der Maschinencode direkt aus dem Syntaxbaum generiert werden kann.

Aufgabe 5: Erzeugung von Maschinencode

Erzeugen Sie den 32 Bit Maschinencode, indem Sie den Syntaxbaum rekursiv durchlaufen. Ein Befehl im Maschinencode ist wie folgt definiert:

```
1 BEFEHL OPERAND_1 OPERAND_2 ...
```

Dabei sind BEFEHL und jeder $OPERAND\_N$ jeweils ein vorzeichenloser 32 Bit Wert. Verwenden Sie zur Kodierung der Befehle den entsprechenden Wert des zur Verfügung gestellten `enum Op` in „ops.h“. Die einzelnen Befehle sind wie folgt definiert:

1. OP_LOOP $END\ STARTX\ STARTY\ STARTZ\ ENDX\ ENDY\ ENDZ$: Dieser Befehl wiederholt den darauf folgenden Codeblock so oft, bis ein internes (nicht sichtbares) Hardware-Register die gewünschten Koordinaten erreicht hat. Der DRAW-, SPHERE-, BOX- und HEART-Befehl nutzt dieses Koordinaten-Hardware-Register ebenfalls, um die aktuellen Koordinaten zu erfahren. Sobald der Schleifenbefehl eine Terminierung der Schleife festgestellt hat, springt er zu dem Wert END in der Datei, die den Maschinencode enthält. Sie können sich END wie eine Zeilennummer vorstellen, nur dass hier der Offset zur ersten 32 Bit Instruktion in der Datei gemeint ist.
2. OP_DRAW REG: Weist die Pseudo-Zielmaschine an, dem Voxel an der Stelle, worauf das Koordinaten-Hardware-Register aktuell zeigt, den Farbwert aus dem Register REG zu zuweisen. Falls der Farbwert invalid ist, wird kein Voxel gezeichnet.
3. OP_JUMP $OFFSET$: Dieser Befehl springt zum $OFFSET$ im Maschinencode, analog zum END im LOOP-Befehl.
4. OP_DISJ $ZIELREGISTER\ LHS\ RHS$: Führt den Disjunktions-Operator | aus. Falls LHS sowie RHS gültige Farbwerte enthalten, wird derjenige von LHS ins $ZIELREGISTER$ geschrieben. Ein ungültiger Farbwert kann nur erzeugt werden, wenn beide Operanden bereits ungültige Farbwerte enthalten.

5. OP_CONJ *ZIELREGISTER LHS RHS*: Führt den Konjunktions-Operator & aus. Falls *LHS* sowie *RHS* gültige Farbwerte enthalten, wird derjenige von *LHS* ins *ZIELREGISTER* geschrieben. Falls einer der Operanden einen ungültigen Farbwert enthält, so ist das Ergebnis ebenfalls ein ungültiger Farbwert.
6. OP_NEG *ZIELREGISTER LHS*: Führt den Negierungs-Operator ! aus. Die daraus neu erzeugten Voxel haben einen schwarzen Farbwert.
7. OP_SET *ZIELREGISTER COLOR*: Setzt das *ZIELREGISTER* auf eine bestimmte konstante Farbe. Sie benötigten diesen Befehl für die Bearbeitung des Praktikums nicht. Sie dürfen ihn aber gerne für Erweiterungen verwenden.
8. OP_SPHERE *ZIELREGISTER X Y Z RADIUS COLOR* Falls sich die aktuelle Koordinate innerhalb der definierte Kugel an der Stelle (X, Y, Z) mit Radius R befindet, wird *COLOR* (andernfalls *INVALID*) ins *ZIELREGISTER* geschrieben.
9. OP_BOX *ZIELREGISTER X Y Z WIDTH HEIGHT DEPTH COLOR* Falls sich die aktuelle Koordinate innerhalb der definierte Box an der Stelle (X, Y, Z) mit der Größe (*WIDTH*, *HEIGHT*, *DEPTH*) befindet, wird *COLOR* (andernfalls *INVALID*) ins *ZIELREGISTER* geschrieben.
10. OP_HEART *ZIELREGISTER X Y Z RADIUS COLOR* Falls sich die aktuelle Koordinate innerhalb des definierten Herzens an der Stelle (X, Y, Z) mit Radius R befindet, wird *COLOR* (andernfalls *INVALID*) ins *ZIELREGISTER* geschrieben.

Sprungbefehle werden in 32-Bit-Schritten innerhalb des Maschinencodes ausgeführt. Bitte beachten Sie also bei ihrer Implementierung, dass ein Befehl auch aus Argumenten zu je 32 Bit bestehen, die ebenfalls übersprungen werden müssen.

ZIELREGISTER, *REG*, *LHS* und *RHS* sind alle 32-Bit-Werte, die jeweils ein Register adressieren, welche einen Farbwert enthalten. Ein ungültiger Farbwert entspricht einem transparenten Voxel und werden daher vom Simulator ignoriert und nicht dargestellt. Sie haben quasi-unendlich viele Register zur Verfügung und können die Register in beliebiger Reihenfolge durchnummerieren.

Wenn die *OFFSET*- bzw. *END*-Sprungmarke jenseits der Länge des Zielcodes ist, terminiert einfachheitshalber das Programm ohne Fehler. Somit ergibt sich eine einfache Möglichkeit eine *LOOP*-Operation terminieren zu lassen, indem man den größtmöglichen 32-Bit-Wert (4294967295) als *END*-Sprungmarke verwendet.

Damit der Maschinencode generiert wird, müssen Sie in der „parser.h“ die Methoden `Fn::code(...)` und `Expr::code(...)` implementieren. Auch in diesem Fall wird die Methode `Expr::code(...)` von `Ast::code(...)` initial aufgerufen. Beachten Sie, dass ein kleiner Teil des zu generierenden Maschinencodes bereits von dieser Methode vorgeben wird. Insbesondere wird erwartet, dass Sie in `Expr::code(...)` das finale Zielregister zurückgeben, welches die zu zeichnende Farbinformation enthält. Denken Sie daran, dass Sie die Opcodes und nicht die Bezeichner in die Datei schreiben. Hierzu verwenden Sie bitte die `opcodes` in „parser.h:18“ sowie `enum Op` in „ops.h“.

Um die korrekten 32-Bit-Werte in die Zieldatei zu schreiben, stellen wir Ihnen ein passendes Interface in der „util.h“ zur Verfügung. Schauen Sie sich als Beispiel die Implementierung von `Ast::code(...)` genauer an, um die Verwendung der Schreibbefehle zu verstehen.

Da der Binärcode nicht für Menschen lesbar ist, haben wir ihnen einen Disassembler (*disasm*) beigelegt, welcher den 32-Bit-Binärcode in eine leichter zu lesende Syntax übersetzt.
Hier ist ein Beispiel, wie es vom Programm (*disasm*) erzeugt wird:

```
#0: LOOP #4294967295 0 0 0 70 70 70
#8: SPHERE r0 50 50 40 15 0xff0000
#15: BOX r1 35 35 40 10 10 10 0x0
#24: NEG r2 r1
#27: CONJ r3 r0 r2
#31: BOX r4 50 50 50 20 20 20 0xff00
#40: HEART r5 20 20 20 10 0xdd0000
#47: BOX r6 0 40 20 24 8 24 0xff
#56: HEART r7 12 41 32 8 0x0
#63: NEG r8 r7
#66: CONJ r9 r6 r8
#70: DISJ r10 r5 r9
```

```
#74: DISJ r11 r4 r10
#78: DISJ r12 r3 r11
#82: DRAW r12
#84: JUMP #0
```

Anhand diesem Beispiels können sie erkennen, dass die Zeilennummern nicht stetig sind. Die Zeilennummern adressieren nämlich den n-ten 32 Bit Wert im Maschinencode, bei dem der Opcode des Befehls steht. Beachten Sie, dass die Zählung informatikergerecht mit null beginnt.

Hinweis: Die genaue implementierte Funktionsweise der einzelne Befehle können sie unter „sim/main.cc“ nachvollziehen.

Fertig!

Herzlichen Glückwunsch! Sie haben nun ihren ersten eigenen Compiler programmiert. Sie können nun mit den folgenden Befehlen den Compiler ausführen und die Zielmaschine simulieren:

Linux:

```
$ ./compiler ../exampleinput.txt ./out.gdi
$ ./sim ./out.gdi ./out.ply
$ meshlab ./out.ply
```

Beachten Sie, dass Meshlab und Co. nicht auf den RBG-Rechnern installiert ist. Sie können alternativ folgende Befehle verwenden, um das Ergebnis für jede Tiefe auf die Konsole zu auszugeben, falls Sie keine Möglichkeit haben ply-Dateien anzuzeigen:

```
$ ./compiler ../exampleinput.txt ./out.gdi
$ ./sim ./out.gdi -
```

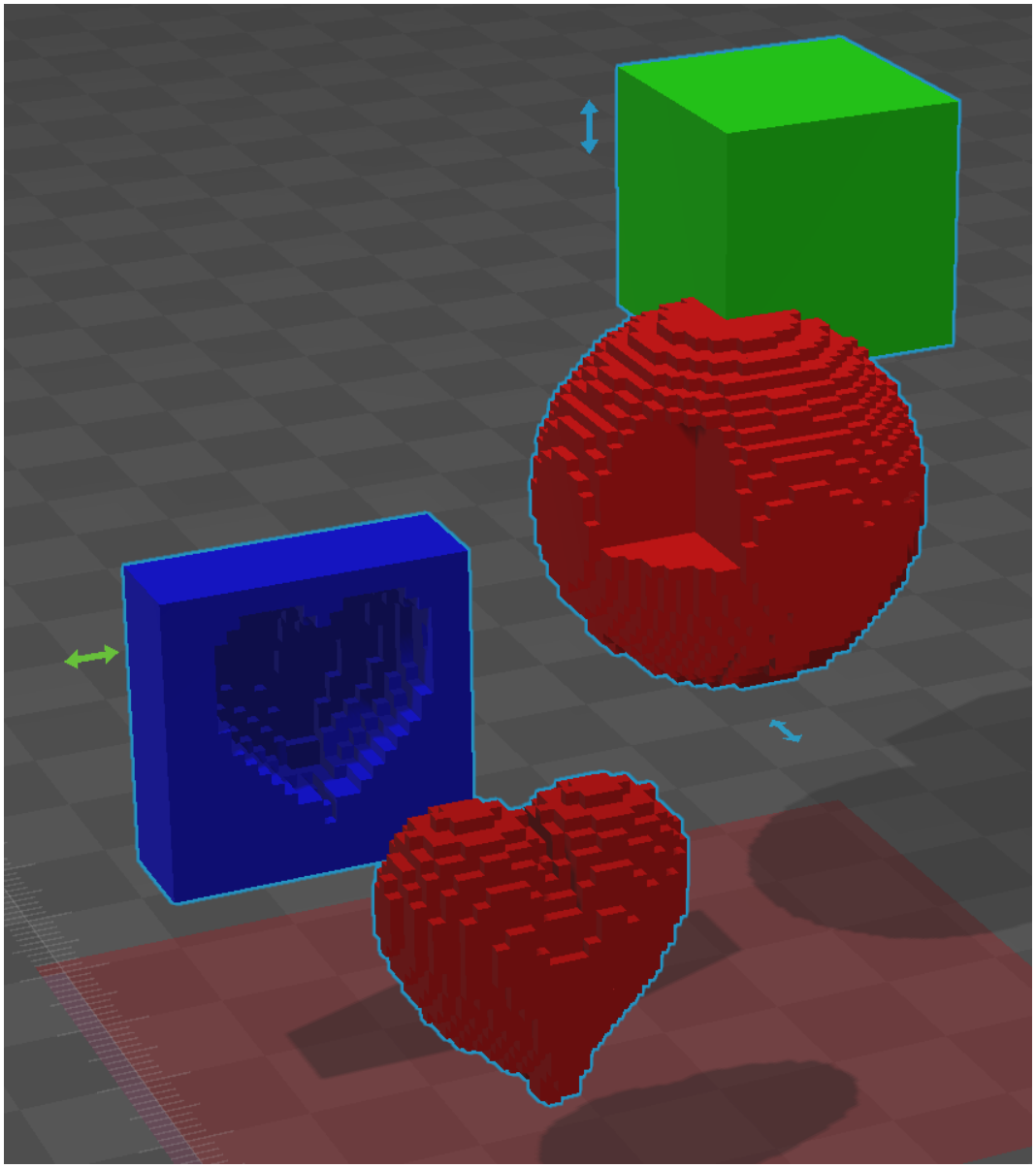
Windows:
Für Windows befinden sich im „build_win“-Ordner analog zu Linux das Script „win_exec.bat“ bzw. „win_exec_noout.bat“.

Die in den Skripten verwendete Datei „exampleinput.txt“ hat folgenden Inhalt:

```
(sphere(50, 50, 40, 15, 0xff0000) & !(box(35, 35, 40, 10, 10, 10, 0x0))) | box(50,
  50, 50, 20, 20, 20, 0x00ff00) | heart(20, 20, 20, 10, 0xdd0000) |

(box(0, 40, 20, 24, 8, 24, 0x0000ff) & !heart(12, 41, 32, 8, 0x0))
```

Falls Ihre Implementierung korrekt ist, dann sollte die ply-Datei folgenden Inhalt darstellen:



Hinweis: Zögern Sie nicht eigene Beispielcodes zu generieren und die daraus resultierende ply-Dateien mit ihren Kommilitonen im Moodle zu teilen! Auf diese Weise können Sie ihre Implementierung gegenseitig vergleichen.