Projektabschlussbericht

Multi-Touch-System
17.12.15

- ## Management- und Dokumentationsattribute

Dokumentationsattribute	
Autor(en)	Raffael Balthasar, Daniel Bannert, Jannis Becker, Thomas Skowronek
Eindeutige Teamnummer	B11
Namen der Teammitglieder	Raffael Balthasar, Daniel Bannert, Jannis Becker, Thomas Skowronek

- ## Problemstellung [T. Skowronek, R. Balthasar]

Durch unser Softwareprodukt, dem Multi-Touch-System, wollen wir folgende Problemstellung lösen. Zur Zeit wird grundlegend jede Software mit einer Computermaus bedient. Durch unser Projekt wollen wir erzielen, dass der Multi-Touch-Tisch die Möglichkeit bietet, aktuelle Software mit den Fingern zu bedienen. Er soll eine möglichst einfache Handhabung bieten mit der sowohl im Arbeitsalltag als auch im Freizeitbereich umgegangen werden kann.
Das dazu entwickelte Billardspiel ist eine Demonstrationssoftware die veranschaulicht, welche Möglichkeiten das Multi-Touch-System bietet. Durch die Touch-Funktionen des Tisches kann man den Queue steuern um so die weiße Kugel anzustoßen.

Um all dies überhaupt zu ermöglichen bedienen wir uns einiger Filter, mit deren Hilfe wir die einzelnen Finger erkennen und in Punkte umwandeln. Anhand dieser Punkte soll es möglich sein, die oben genannte Software zu bedienen. Störungen allgemein, z.B. der Handballen oder Rauschen sollen herausgefiltert werden.

- ## Abweichungen zum Pflichtenheft [D. Bannert]

Visionen und Ziele

/PZ20/ Verschiedene interaktive Demonstrationssoftwares zur Veranschaulichung der Fingererkennung durch die Kamera.

Diese Vision beinhaltete die Planung einer 3D-Bowlingsimulation sowie einer 2D-Zeichensoftware. Aus zeitlichen Gründen konnte wir letztere nicht mehr in Angriff nehmen wodurch wir uns einzig auf die 3D-Simulation konzentrierten. Diese stellte sich durch die komplexe Physikberechnung von umherfliegenden und sich gegenseitig wegstoßenden Kegeln aufgrund der Formen allerdings als äußert kompliziert heraus.
Um dieses Problem zu umgehen einigten wir uns kurzfristig auf die Erstellung einer Billardsimulation bei der die Physik zwar immer noch kompliziert, aber aufgrund der gleichmäßigen Beschaffenheit einer Kugel sowie deren Schwerpunkt in der Mitte, einfacher

umzusetzen war. Dies soll und wird den Erfahrungswert, welchen der Touch-Tisch bietet, in keinster Weise schmälern.

/PZ20 Version 2/: Eine Billardsimulation mit 2 Spielern und eigens entwickelter Physikberechnung sowie drehbarer Kamera zur Veranschaulichung der Funktion des Mutli-Touch-Tisches.

Funktionale Anforderungen

/PF40/ Die verschiedenen in JoGL implementierten Demos MÜSSEN korrekt funktionieren

Aus den oben dargestellten Gründen musste diese funktionale Anforderung umgeschrieben werden. Einige weitere wurden zusätzlich ergänzt um den gesamten Umfang der Billardsimulation abzudecken.

/PF40 Version 2/ Die in JoGL implementierte Billardsimulation MUSS korrekt funktionieren und durch die Fingererkennung der Bildverarbeitung spielbar sein

/PF60 (neu)/ Die Billardsimulation MUSS von bis zu zwei Spielern spielbar sein

/PF70 (neu)/ Die Billardsimulation MUSS eine Spielfläche, Wände zur Abgrenzung, 6 Löcher und insgesamt 16 Kugeln beinhalten.

/PF80 (neu)/ Zu den 16 Kugeln MÜSSEN regelkonform 7 pro Farbe sowie eine schwarze und eine weiße gehören

/PF90 (neu)/ Die Kugeln MÜSSEN animiert sein, sich auf dem Spielfeld also bewegen können

/PF100 (neu)/ Das Verhalten einer Kugel beim Auftreffen auf ein Hindernis (eine andere Kugel oder eine Wand) MUSS realitätsnah umgesetzt sein

/PF110 (neu)/ Die Kamera im Spiel muss vom Spieler drehbar sein

• Verwendete Technologie [J. Becker]

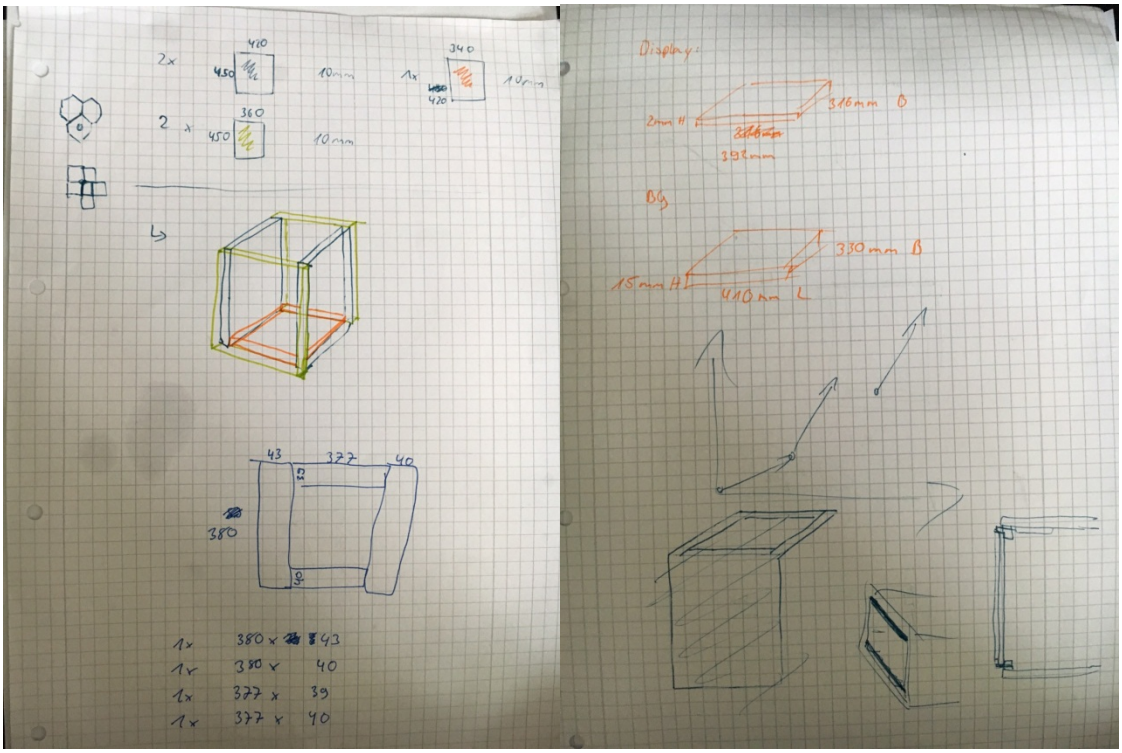
Bildverarbeitung: OpenCV 2.4.11, 17" TFT-Monitor, PlayStation-Eye (Kamera), Acrylglasplatte, Infrarot-LEDs, IR-Passfilter (Diskette)

Computergrafik: Java JDK8, JoGL 2.3.2, Vecmath 1.5.2, Violet 2.0 (UML-Diagramme), ObjectAid als Plugin für Eclipse (Klassendiagramm)

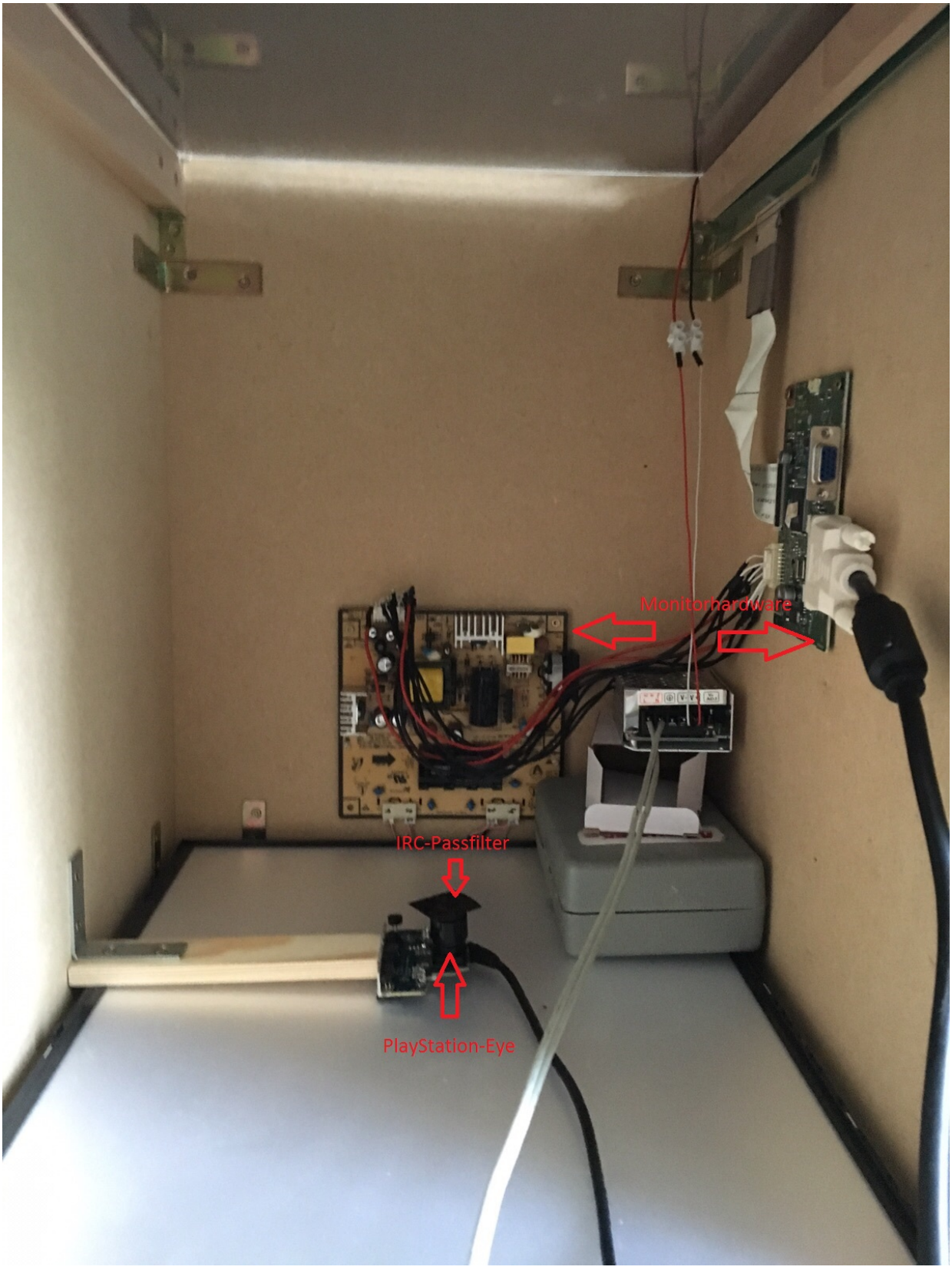
- **Lösungsstrategie [D. Bannert, R. Balthasar, T. Skowronek, J. Becker]**

Hardware

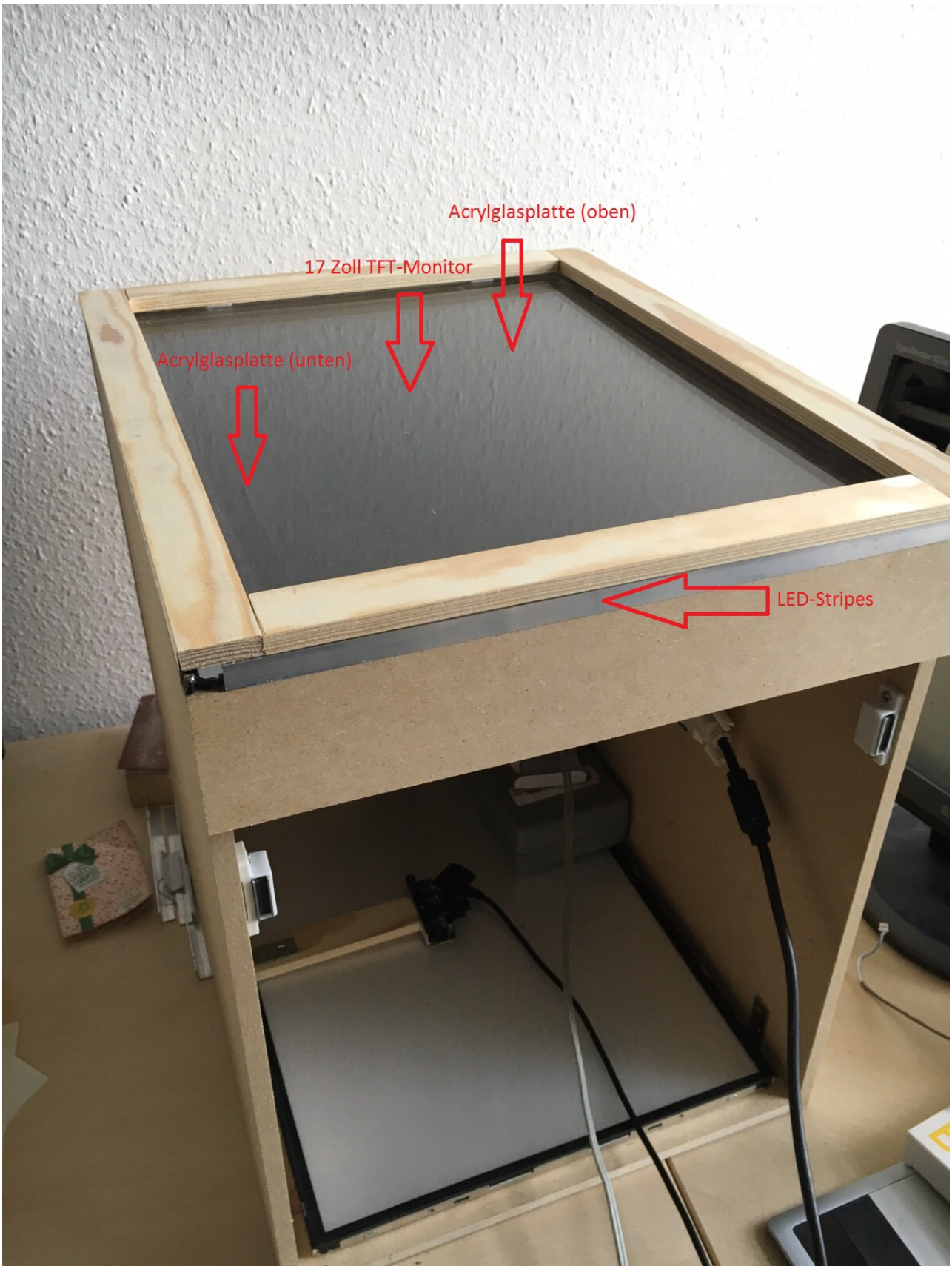
Zur Herstellung des Touch-Tisches verwendeten wir gewöhnliche Holzbalken als Grundgerüst welches dann später mit wichtiger Hardware gefüllt wurde.
Die spätere Erkennung der Finger musste hier in die Wahl der Hardware mit einbezogen werden, da diese durch Filterung von Infrarot-Strahlen erfolgen sollte. Zur Erstellung jener Infrarot-Strahlen verwenden wir Infrarot-Stripes welche viele kleine Infrarotlampen enthalten. Diese Stripes wurden rings um eine Acrylglasplatte befestigt. In dieser Acrylglasplatte werden die Infrarot-Strahlen in alle Richtungen reflektiert und treffen so auch auf die zentral darunter liegende Kamera.
Ein PlayStation-Eye als Kamera mit leicht zu entfernendem Infrarotblockfilter sowie ein darüber liegender IR-Passfilter einer Diskette sorgen dafür, dass ausschließlich Infrarot-Strahlen erkannt werden.
Unter der Acrylglasplatte liegt ein 17“ TFT-Monitor, welcher von einer weiteren Acrylglasplatte unterhalb des Bildschirms befestigt wird. Die Hardware des Monitors ist innerhalb des Holzkastens befestigt.



Zur genauen Planung des Aufbaus verwendeten wir viele Skizzen. Dieses Vorgehen zog sich später auch durch den Computergrafik-Teil um das Physikverhalten einfacher nachzuvollziehen.



Das Innenleben des Touch-Tisches bestehend aus der Kamera, einem IRC-Passfilter einer Diskette und der Monitorhardware. Oben am Rand erkennt man die untere Acrylglasplatte.



Der Tisch von außen. Die silbernen LED-Leisten beleuchten die Acrylglasplatte. Zwischen den beiden Platten befindet sich der 17-Zoll TFT-Monitor.

Bildverarbeitung

Im Teil der Bildverarbeitung werden nun die auf die Kamera treffenden Infrarot-Strahlen ausgewertet. Liegt ein Objekt auf der Acrylglasplatte so werden eintreffende Infrarot-Strahlen in Richtung der Kamera reflektiert und dort von dieser erkannt. Auf dem Bildschirm werden solche Stellen durch einen Grauwertfilter weiß bzw. hellgrau dargestellt. Je näher das Objekt desto heller erscheint der Punkt auf dem Bildschirm.
Durch die Reflektion der Decke in einem Raum wird auch diese durch den Algorithmus erkannt, wenn schwach dank der Entfernung zur Kamera. Um dieses Unerwünschte Extra zu entfernen wird das 5. Bild welches die Kamera aufnimmt als Hintergrundbild gesetzt und später von jedem neuen Bild abgezogen, wodurch der Hintergrund entfernt und somit gänzlich schwarz wird. Wir verwenden das 5. Bild da die ersten Bilder eventuelle Störungen seitens der Kamera enthalten könnten.
Nun können die Finger aufgelegt werden. Die direkt aufliegenden Fingerspitzen werden nun als recht starke, hellgraue bis weiße Punkte dargestellt. Der Handballen im Hintergrund jedoch wird auch noch leicht zu sehen sein. Um dem zu entgehen verwenden wir einen Hochpassfilter der nur starke hohe Frequenzen durchlässt. Die Handballen oder etwaige zusätzliche Gegenstände im Hintergrund wie Armbänder verschwinden so von der Bildfläche. Ein Unschärfefilter entfernt zudem etwaiges Rauschen im Bild.
Nun sind nur noch die Fingerspitzen als weiße bzw. hellgraue Punkte zu erkennen. Um diese jedoch ohne Probleme zu verwenden nutzen wir eine Multiplikation sowie einen weiteren Hochpassfilter, dieses Mal einen binären Hochpassfilter, der für eine eindeutige Unterteilung in schwarz und weiß sorgt. Mit diesen starken, weißen Punkten können wir nun arbeiten. Die Punkte werden durch einen entsprechenden Algorithmus erkannt und die Fingerbewegung ersetzt die Maus.
Zur Kalibrierung verwenden wir eine eigene Klasse. Diese überprüft die exakten Koordinaten der Eckpunkte.

Die Filter haben wir aus Zeit- und Performancegründen nicht selbst geschrieben sondern mit Hilfe von OpenCV implementiert.

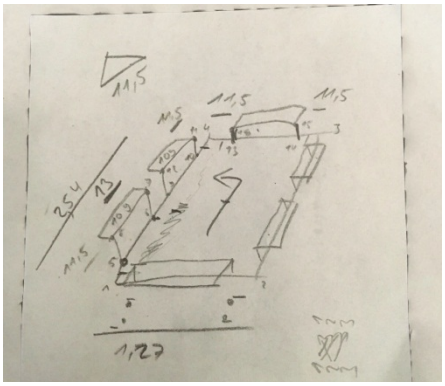
Computergrafik

Im Teil der Computergrafik galt es nun eine Demonstrationssoftware für den Multi-Touch-Tisch zu erstellen. Für die Modellierung verwendeten wir OpenGL für Java. Geplant waren 3D- sowie 2D-Software wobei die Erstellung der 2D-Software aus Gründen der Aufgabenstellung und zu Gunsten einer besseren 3D-Software zurückgestellt wurde.
Nach einiger Einarbeitungszeit in JoGL gelangen uns erste Erfolge in der Polygonerstellung. Bevor wir jedoch mit der Erstellung der Bowlingbahn, Kegel und Kugel begannen machten wir uns die Funktionsweise anhand von 2D-Polygonen in Form von Dreiecken und Vierecken klar und erstellten einen sogenannten Vertexbuffer um schnell und simpel Polygone erstellen zu können. Durch den Vertexbuffer werden zudem sämtliche Vertices direkt auf die Grafikkarte geschoben wodurch die Performance merklich verbessert wird. Nach Erstellung der 3D-Plane für die Bowlingbahn testeten wir die Rotationsmethoden von OpenGL die später für die Kugel benötigt werden.
Die Kugel selbst jedoch stellte sich als unser bisher größtes Problem dar. Da der Vertexbuffer sämtliche Informationen über jeden Vertex benötigt muss dementsprechend auch jeder Vertex angegeben werden. Eine Kugel jedoch besteht aus abertausenden von Vertices sofern sie Rund sein soll.

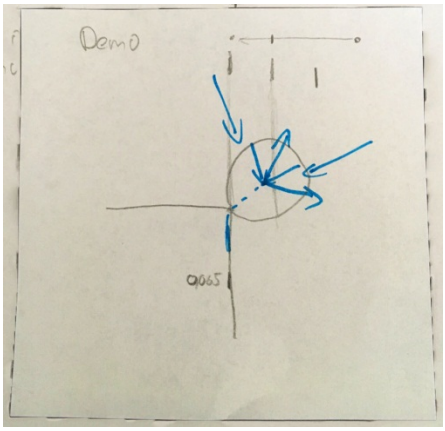
Nachdem uns die Kugel gelang stellten wir jedoch fest, dass sich die Physik bei der Kollision zwischen Kugel und Kegel sowie zwischen den jeweiligen Kegeln zu umfangreich und zeitaufwändig gestaltete. Wir entschieden uns also stattdessen für ein Billardspiel. Die Kollisionen zwischen einzelnen Kugeln und zwischen den Kugeln und Wänden ließen sich aufgrund der Beschaffenheit der Spielkugeln deutlich einfacher berechnen.

Das Billardspiel sollte recht einfach gestaltet werden damit die verbliebende Zeit für die komplexe Physik verwendet werden konnte. Dies gelang uns indem wir das Spielfeld durch eine einfache Plane darstellten. Die Wände sind zur Hauptplane orthogonal platzierte Flächen an den Rändern. Lücken in diesen Flächen symbolisieren die Löcher des Billardtisches.
Die Maße der Spielfläche richten sich nach den offiziellen Auflagen für Billardtische. Der Durchmesser der Kugeln ist ein kleines bisschen größer also vorgegeben, da uns eine glatte Zahl (6cm) die Positionierung auf dem Spielfeld erleichterte.
Die Kugeln (16 Stück an der Zahl) wurden nicht mit Texturen versehen sondern einfach in blaue, rote, schwarze und weiße Kugeln unterteilt wobei es 7 blaue, 7 rote, eine schwarze und eine weiße Kugel gibt.
Diese Kugeln erhielten einen Bewegungsvektor der sich je nach Situation anpasst.
Nachdem die Kugeln auf dem Spielfeld fachgerecht positioniert wurden implementierten wir die Physik und testeten diese durch ein automatisches Anstoßen der weißen Kugel bei Programmstart mit unterschiedlichen Geschwindigkeiten. Die Kugeln flogen tatsächlich in alle Richtungen und schienen sich realitätsnah zu verhalten. Bewerkstelligt wurde die Kollisionsabfrage durch eine Koordinatenabfrage der Wand bzw. die Beträge dieser Koordinaten. Sollte eine Kugel an irgendeinem Punkt ihrer Oberfläche (also alle Punkte mit 3cm(Radius der Kugel) vom Mittelpunkt entfernt) an eine Wandkoordinate gelangen so wird der Bewegungsvektor der Kugel einfach umgelenkt.

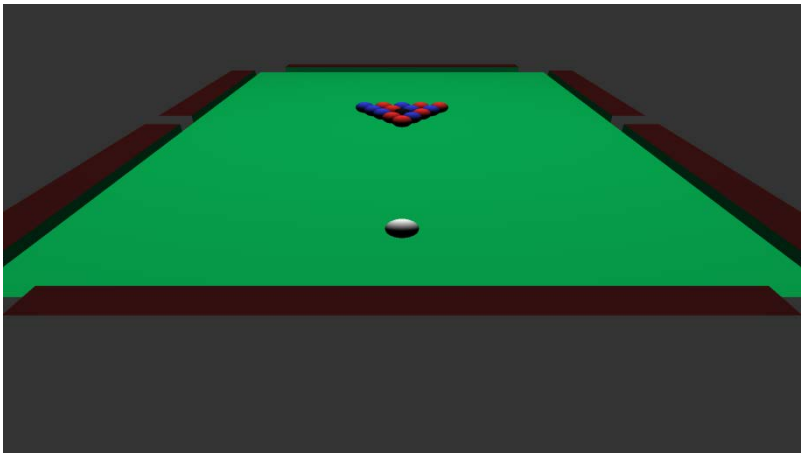
Bei zu hoher Geschwindigkeit (die allerdings unrealistisch war) wurden jedoch einige Frames übersprungen, was dazu führte dass die Kugeln durch die Wände glitten und die Physikberechnungen erst hinter der Wand in Aktion getreten sind. Dies führte zu weiteren unschönen Bugs die den Spielfluss störten. Eine Anpassung der Framerate sowie eine Optimierung der Physik ließen dieses Problem verschwinden. Da die Kugeln beispielsweise an der rechten Seite des Feldes bei Wandkontakt nur in linker Richtung abprallen können bauten wir diese Einschränkung in den Code ein und umgingen so das Problem, dass Kugeln hinter den Wänden von Physik betroffen waren.
Prallte eine Kugel jedoch mit einem kleinen Stück des Körpers an eine Kante wodurch sie dennoch ins Loch fallen sollte so wurde die Kugel stattdessen von der Wand abgestoßen. Dieses Problem lösten wir durch genauere Feinjustierung der Physik sowie der Einschränkung dass Kugeln die mit maximal 33% ihres Körpers an eine Ecke stoßen (die jeweiligen Ecken wurden durch fixe Koordinatenangaben festgelegt), so verhalten sie sich anders und prallen ins Loch um eine realitätsnahe Physik zu gewährleisten.
Beim Aufprall einer Kugel auf ein Hindernis wird durch eine einfache Verringerung des Bewegungsvektors die Geschwindigkeit gedrosselt. Zudem verliert eine ungehinderte Kugel nach und nach an Geschwindigkeit.
Da die Geschwindigkeitsreduzierung einem Faktor unterliegt kann eine Kugel niemals wirklich zum Stillstand kommen, jedoch ist die Geschwindigkeit nach kurzer Zeit so klein dass das menschliche Auge diese Bewegung niemals erkennen könnte. Dies ist später wichtig um das Ende einer Runde zu erkennen.
Wenn eine Kugel „in“ ein Loch fiel rollte es stattdessen einfach gerade aus dem Spielfeld. Um das Fallen der Kugeln zu symbolisieren veränderten wir beim Verlassen des Spielfeldes mit mindestens 2/3 des Kugelumfangs die y-Koordinate des Bewegungsvektors. Nach einer Fallzeit von einer Sekunde wird die Kugel dann nicht mehr gerendert und verschwindet. Bereits beim Verlassen des Spielfeldes wird sie aus der ArrayList, welche die Kugeln im Feld umfasst, entfernt und dadurch als eingelocht eingestuft. So können wir später erkennen welcher Spieler welche Kugel versenkt hat und welche Auswirkungen dies auf das weitere Spielgeschehen hat (Schwarze Kugel zu früh eingelocht -> Spiel verloren. Kugel falscher Farbe eingelocht -> Foul, der nächste Spieler ist an der Reihe).
Um eine einfache Bedienung des Spieles zu gewährleisten lässt sich die Blickperspektive bzw. Kamera durch einfache Fingerbewegungen drehen. Diese Drehung geschieht um die weiße Kugel. Eine schnelle Bewegung mit dem Finger in Richtung der weißen Kugel stößt diese an. Die Geschwindigkeit der Kugel wird aus der Geschwindigkeit der Fingerbewegung berechnet. Während die Kugeln in Bewegung sind kann der Spieler keine weiteren „Queuebewegungen“ ausführen. Währenddessen dreht sich die Kamera bei Fingerbewegungen zur besseren Übersicht um den Mittelpunkt des Spielfeldes.
Um darzustellen welcher Spieler an der Reihe ist oder wer gewonnen hat wollten wir mit Hilfe von OpenGL Text im Spielfenster rendern. Dies funktionierte jedoch nicht wie geplant weshalb wir aus Zeitgründen auf die Analyse des Problems verzichteten und stattdessen die System Out Ausgabe verwenden.

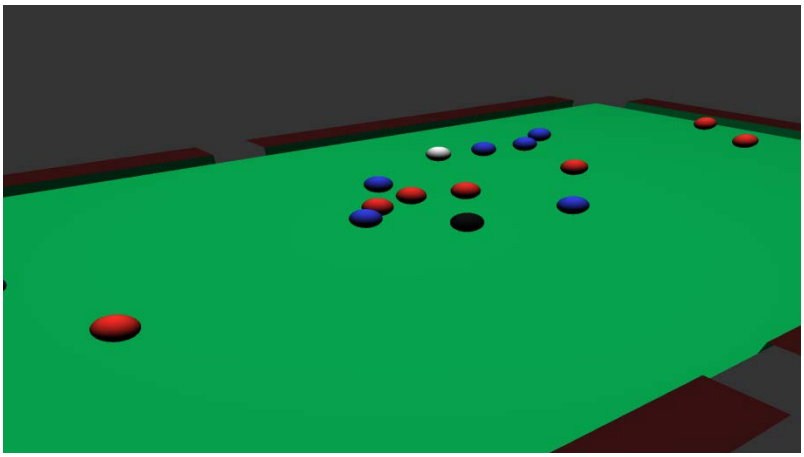


Skizze für die Bemaßung der Spielfläche und Wände



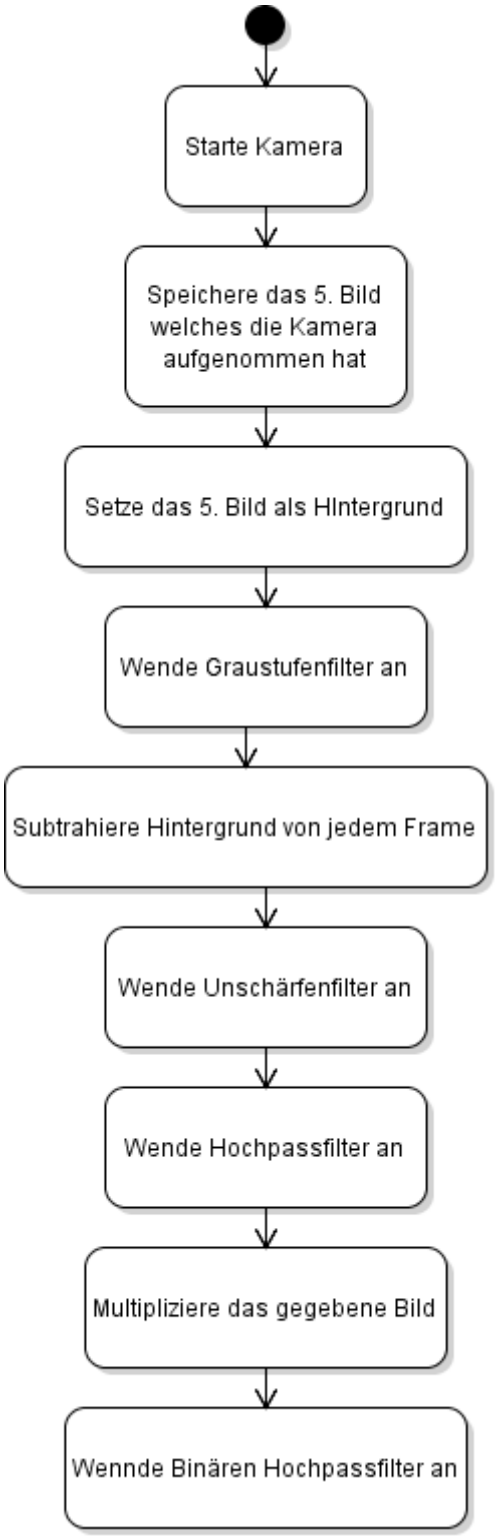
Skizze für das Auftreffverhalten einer Kugel


Ingame Screenshot bei Spielstart

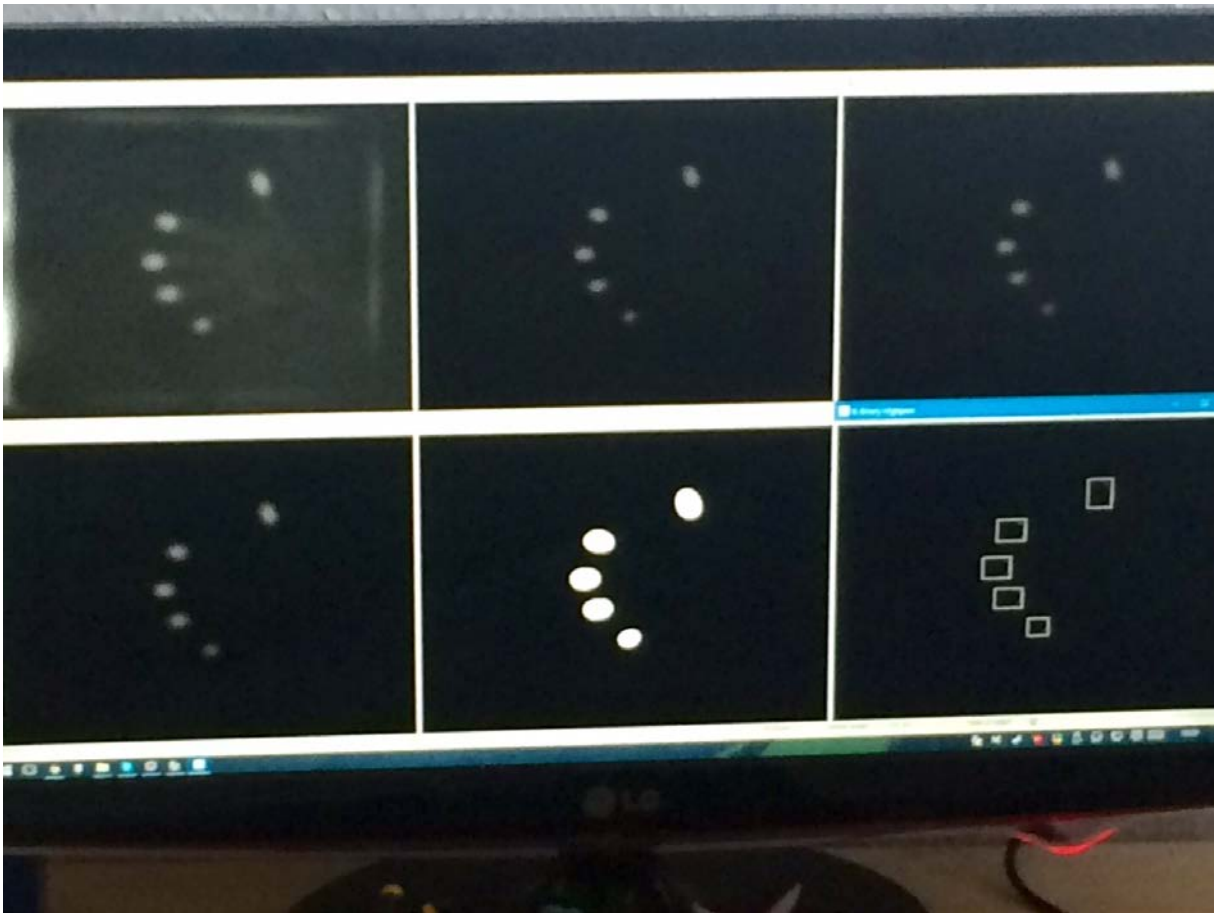

Screenshot während des Spiels

- **Algorithmus [R. Balthasar, D. Bannert, T. Skowronek, J. Becker]**

Bildverarbeitung – Filteranwendungen



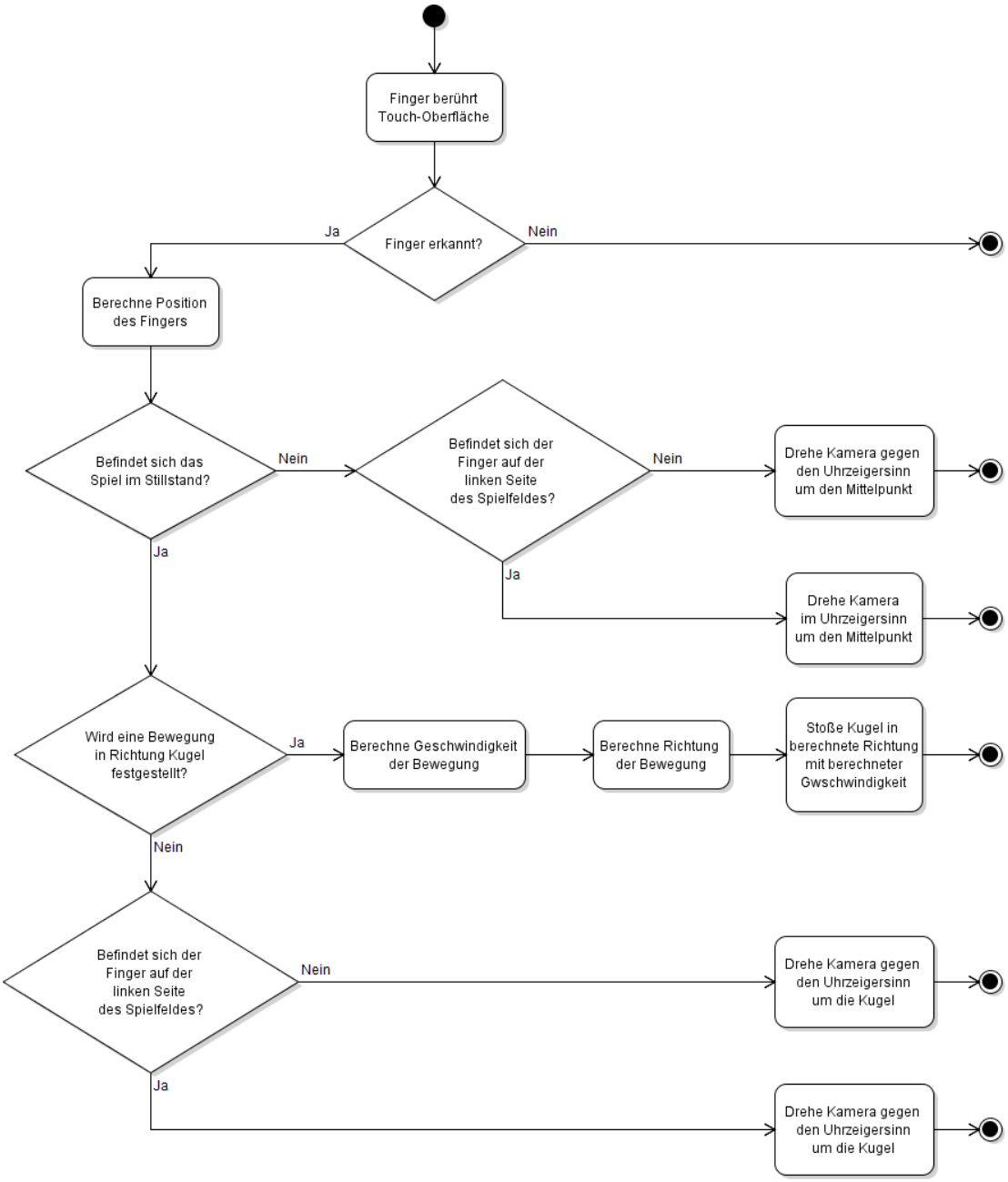
Dieses Aktivitätsdiagramm zeigt die verschiedenen Filter welche in unserem Programm verwendet werden um die Fingererkennung durchführen zu können. Die Kamera nimmt direkt nach ihrer Aktivierung auf. Dabei können jedoch bei den ersten Bildern einige Artefakte entstehen weshalb wir vorsichtshalber das 5. Aufgenommene Bild der Kamera als Hintergrund verwenden. Im Anschluss werden sämtliche Farben in Grauwerte umgewandelt. Nun wird das Hintergrund von jedem neuen Bild, welches die Kamera aufnimmt, subtrahiert. Dies entfernt sämtliche im Hintergrund sichtbare Gegenstände und Wände. Es entsteht ein schwarzes Bild. Wenn nun Finger auf die Bildschirmfläche gelegt werden so sind diese in hellgrau – weiß zu erkennen. Ein Unschärfefilter sorgt für vermindertes Rauschen im Bild während der erste Hochpassfilter nur starke Frequenzen passieren lässt. Dadurch verschwinden beispielsweise „Geisterhände“, verursacht durch Handballen die sich leicht über der Bildschirmfläche befinden. Eine Multiplikation verstärkt die Frequenzen noch einmal zur erleichterten Erkennung. Der abschließende binäre Hochpassfilter sorgt dann noch für eine absolute Unterteilung in Schwarz und Weiß. Die nun weiß dargestellten „Fingerabdrücke“ können nun erkannt und verwendet werden.



Die verschiedenen Filteranwendungen kurz zusammengefasst:

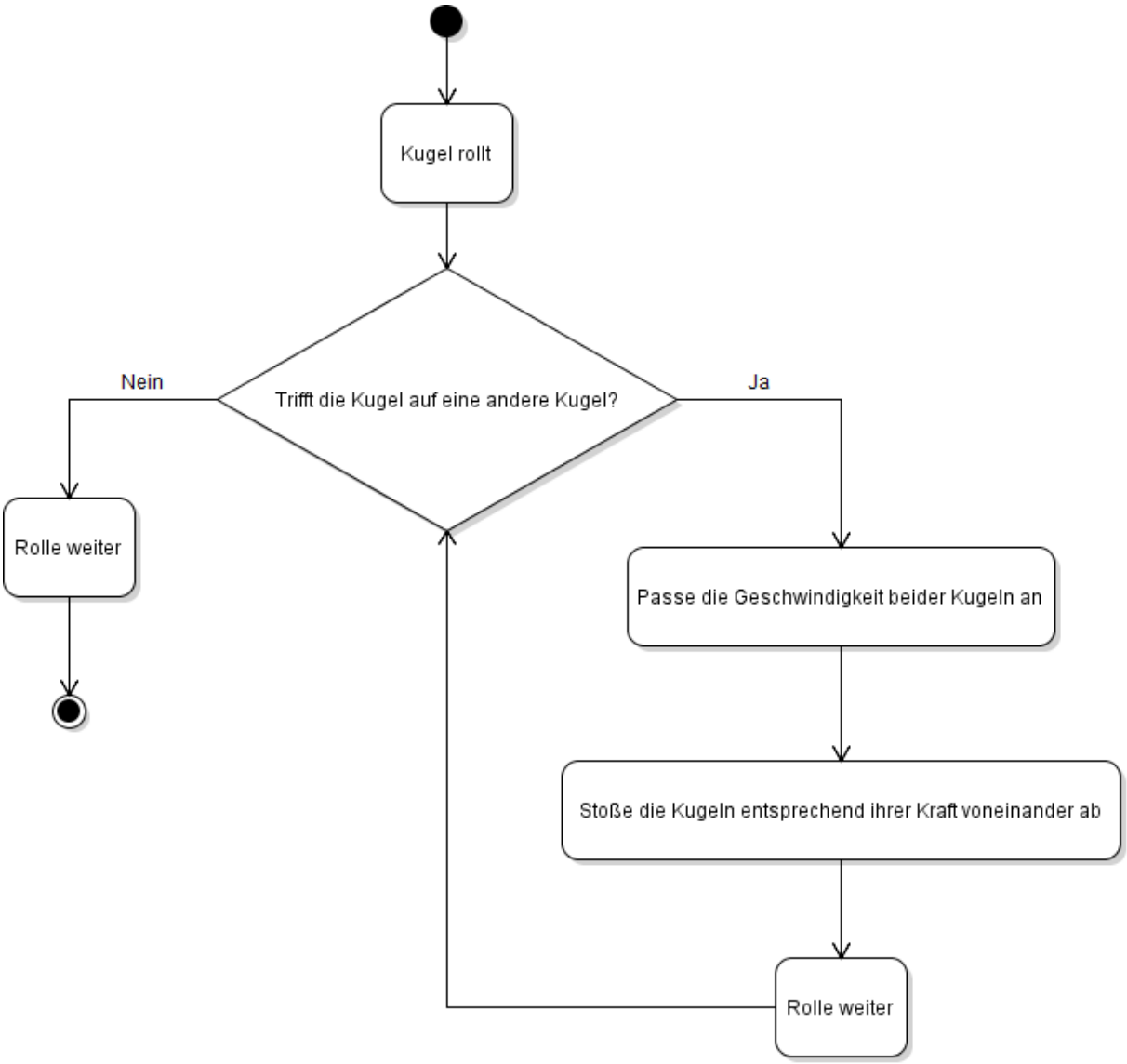
Graustufenfilter, Hintergrund Subtraktion, Unschärfenfilter, Hochpassfilter, Multiplikation, Binärer Hochpassfilter (im letzten Bild existieren die Rechtecke nur zur Verdeutlichung der Punkterkennung)

Bildverarbeitung und Computergrafik - Kamerabewegung



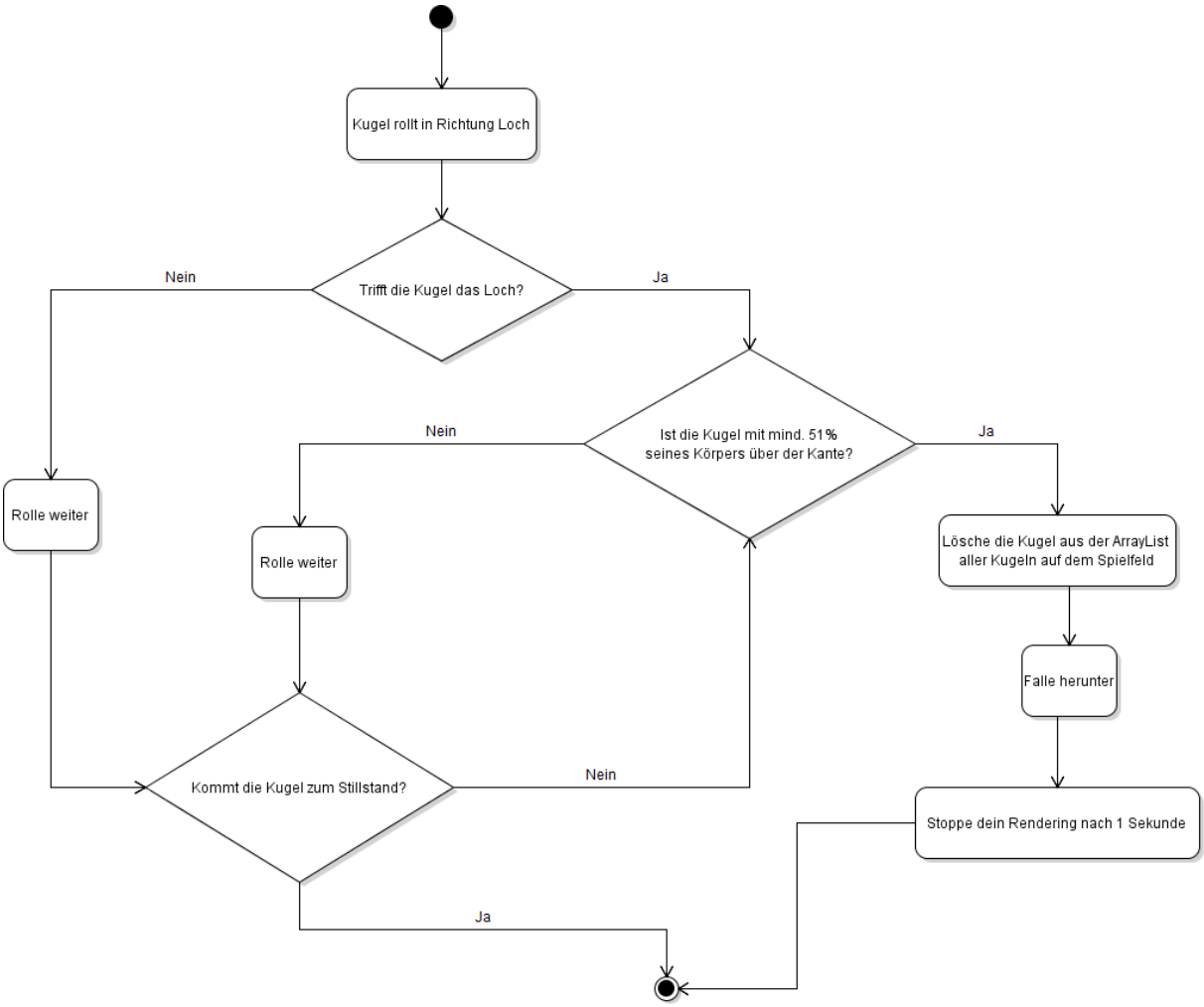
Dieser Algorithmus befasst sich mit der Kamerabewegung im Spiel mit Hilfe der Fingerbewegungen. Berührt ein Finger die Oberfläche des Touch-Tisches so wird seine Position erkannt und verwendet. Falls sich das Spiel gerade in Bewegung befindet, die Kugeln also nicht still stehen, so wird abgefragt ob sich der Finger auf der Oberfläche befindet. Befindet er sich auf der linken Hälfte des Bildschirmes so wird die Kamera gegen den Uhrzeigersinn um den Mittelpunkt des Spielfeldes gedreht. Befindet er sich auf der rechten Hälfte so dreht sich die Kamera entsprechend im Uhrzeigersinn. Befindet sich das Spiel jedoch im Stilstand so wird die Kamera je nach entsprechender Fingerposition um die weiße Kugel gedreht. Eine schnelle Bewegung in Richtung der Kugel stößt diese an. Die Geschwindigkeit und die Richtung der Bewegung zum Anstoßen der weißen Kugel werden zur Berechnung der Geschwindigkeit und Richtung der Kugel selbst verwendet. Die Kugel setzt sich danach in Bewegung.

Computergrafik - Kugelphysik Teil 1: Kugel trifft auf Kugel



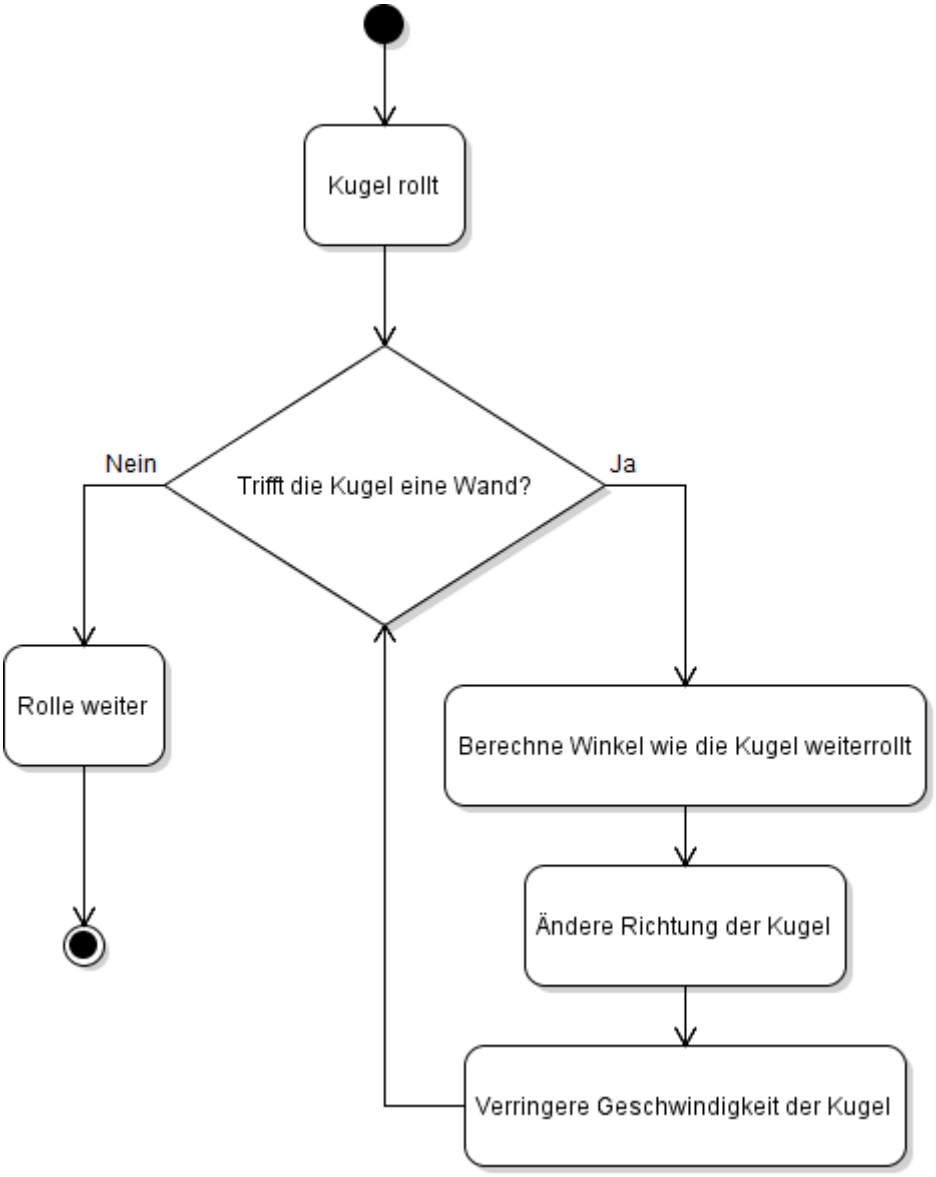
Dieser (hier vereinfacht dargestellte) Algorithmus sorgt für die Grundphysik zwischen den sich bewegenden Kugeln. Jede Kugel wird durch Reibung von sich aus dauerhaft abgebremst bis sie schließlich zu einem Zustand gelangt der für das menschliche Auge dem Stillstand gleichkommt.
Trifft eine Kugel jedoch auf eine andere Kugel so wirkt die Physik. Die Geschwindigkeiten der Kugeln werden angepasst. Trifft beispielsweise eine schnelle Kugel auf eine stehende oder langsam vorher rollende Kugel so wird die getroffene Kugel beschleunigt während die treffende Kugel verlangsamt wird und von der getroffenen Kugel je nach vorheriger Geschwindigkeit abprallt oder aber langsamer hinterherrollt. Treffen sich zwei sich bewegende Kugeln während beide aufeinander zu rollen so stoßen sie sich gegenseitig in die jeweils andere Richtung und werden verlangsamt.
Die Abfrage ob eine Kugel auf eine andere trifft wird jeden Frame ausgeführt.

Kugelphysik Teil 2: Kugel trifft auf Loch



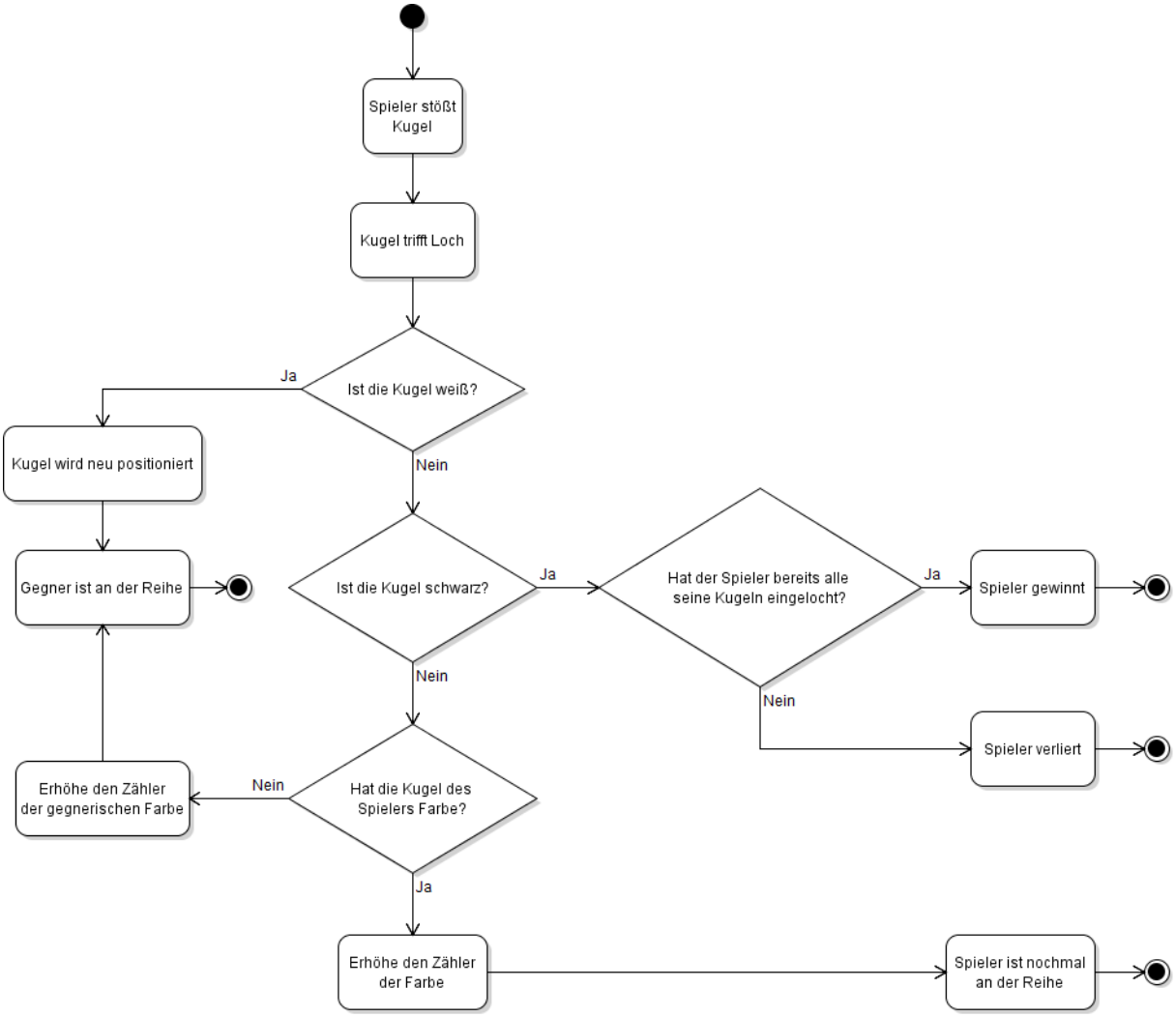
Dieser Algorithmus behandelt den Fall des Einlochens einer Kugel. Die Löcher werden durch Lücken in den Wänden des Spielfeldes dargestellt und lassen Kugeln dementsprechend an diesen Koordinaten passieren.
Landet nun eine Kugel bei einem besagten Loch so wird überprüft ob sich mindestens 51% ihres Körpers über dem Loch befinden. Ist dies nicht der Fall so rollt die Kugel unberührt weiter oder kommt zum Stillstand, je nachdem wie langsam sie vorher war.
Ist jedoch der Großteil des Körpers über dem Loch so fällt die Kugel herunter. Dies wird durch die Veränderung der y-Koordinate des Bewegungsvektors bewerkstelligt. Gleichzeitig wird die Kugel aus der ArrayList aller bestehenden Kugeln auf dem Spielfeld gelöscht. Die Kugel selbst wird nach einer Fallzeit von einer Sekunde zudem nicht mehr gerendert.
Durch das Löschen der Kugel aus der ArrayList kann nun überprüft werden ob ein Spieler bereits alle seine Kugeln eingelocht hat. Da dies einen wichtigen Einfluss auf das Spielgeschehen hat wird dies zu einem späteren Zeitpunkt im Projektbericht noch genauer erläutert.

Kugelphysik Teil 3: Kugel trifft auf Wand



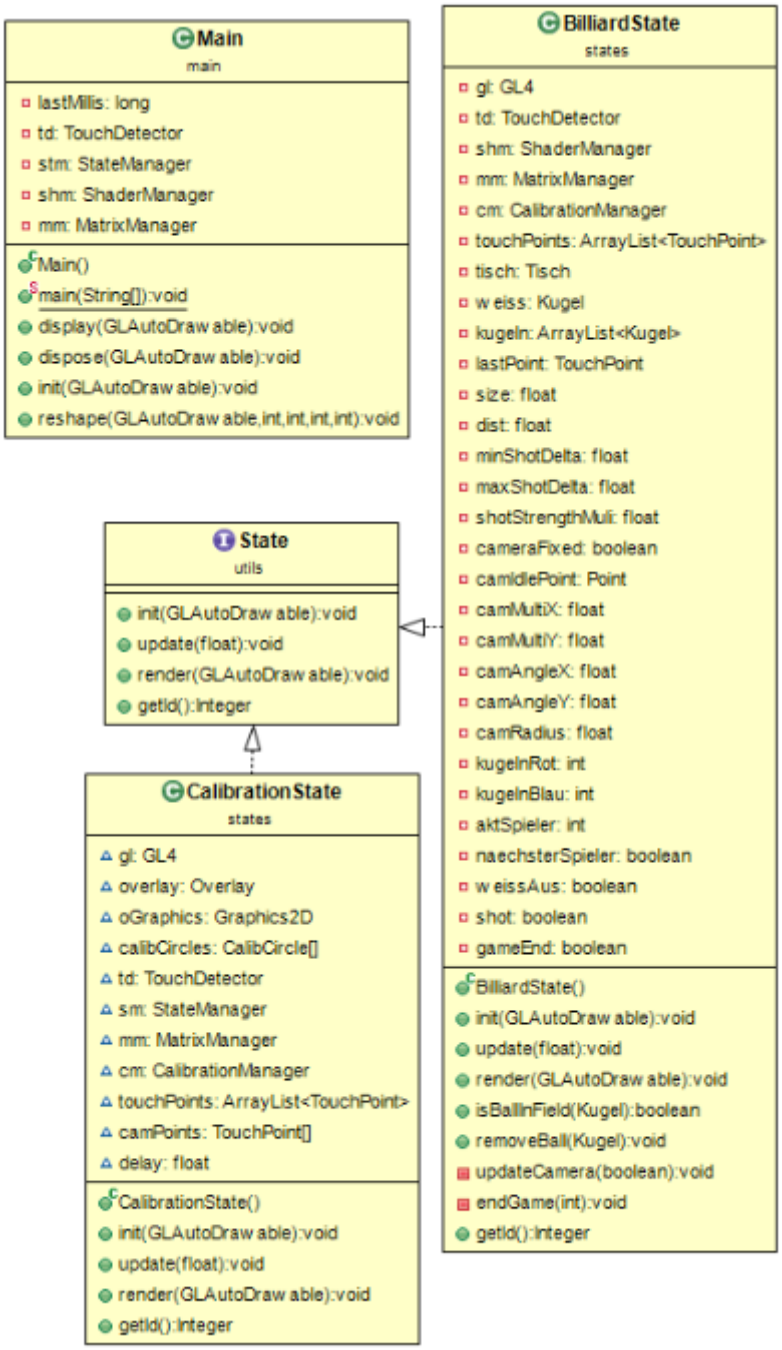
Dieser Algorithmus tritt in Aktion wenn eine Kugel eine Wand berührt und somit von ihr abgelenkt werden muss. Da die Wände in Koordinaten angegeben sind lässt sich ziemlich einfach berechnen ob eine Kugel mit dem Radius 3cm eine Wand an der jeweiligen Koordinate berührt. Da die Koordinaten der Wandlücken aus diesem Algorithmus ausgelassen wurden rollen die Kugeln im Falle eines Loches weiter (siehe Kugelphysik Teil 2). Trifft die Kugel in ihrer Bewegung auf keine Wand so rollt sie einfach weiter und unterliegt, wie vorher bereits beschrieben, dem Geschwindigkeitsverlust durch Reibung.
Trifft die Kugel jedoch auf eine Wand so wird der Eintritts- sowie Austrittswinkel berechnet und die Richtung der Kugel sowie dessen Geschwindigkeit geändert. Die Kugel rollt nun wie gewohnt, allerdings langsamer als zuvor, weiter.

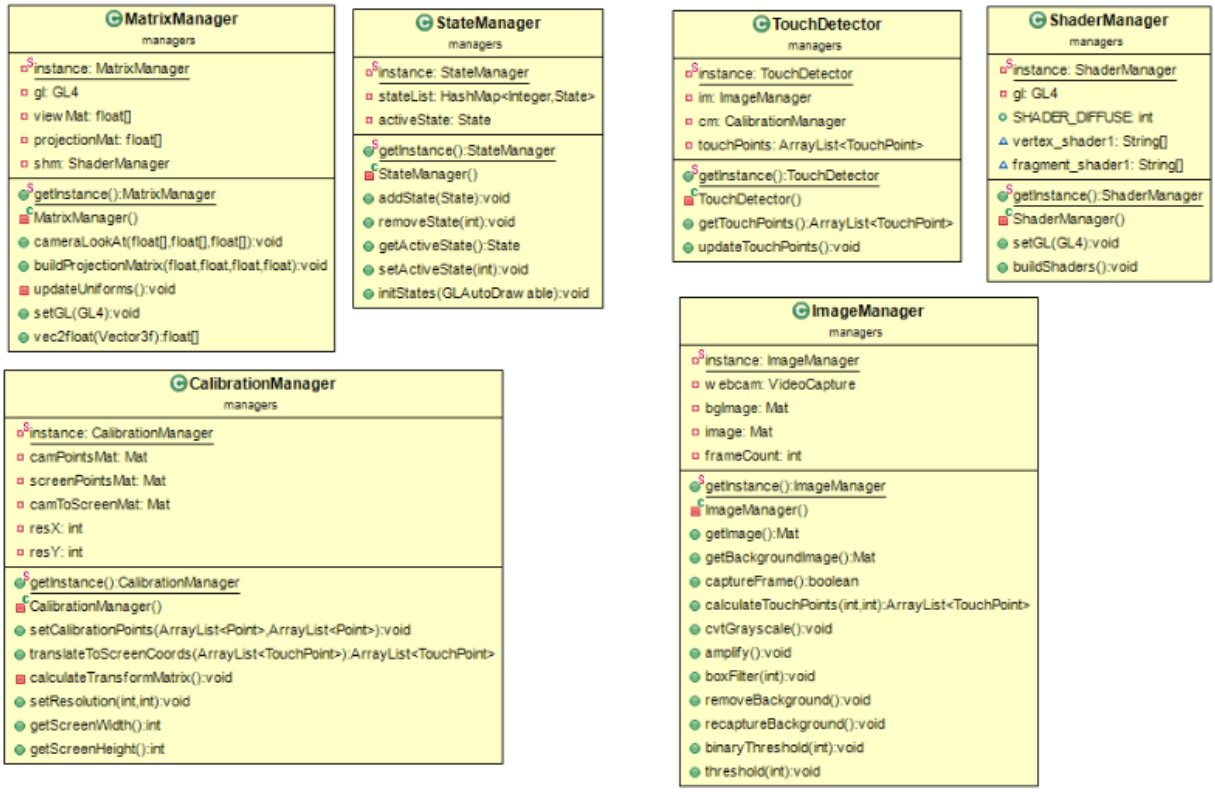
Regelwerk: Kugel geht ins Loch

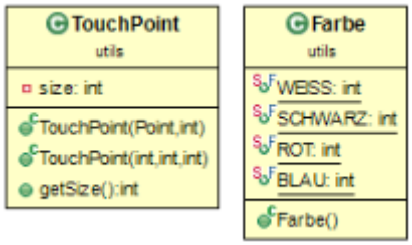


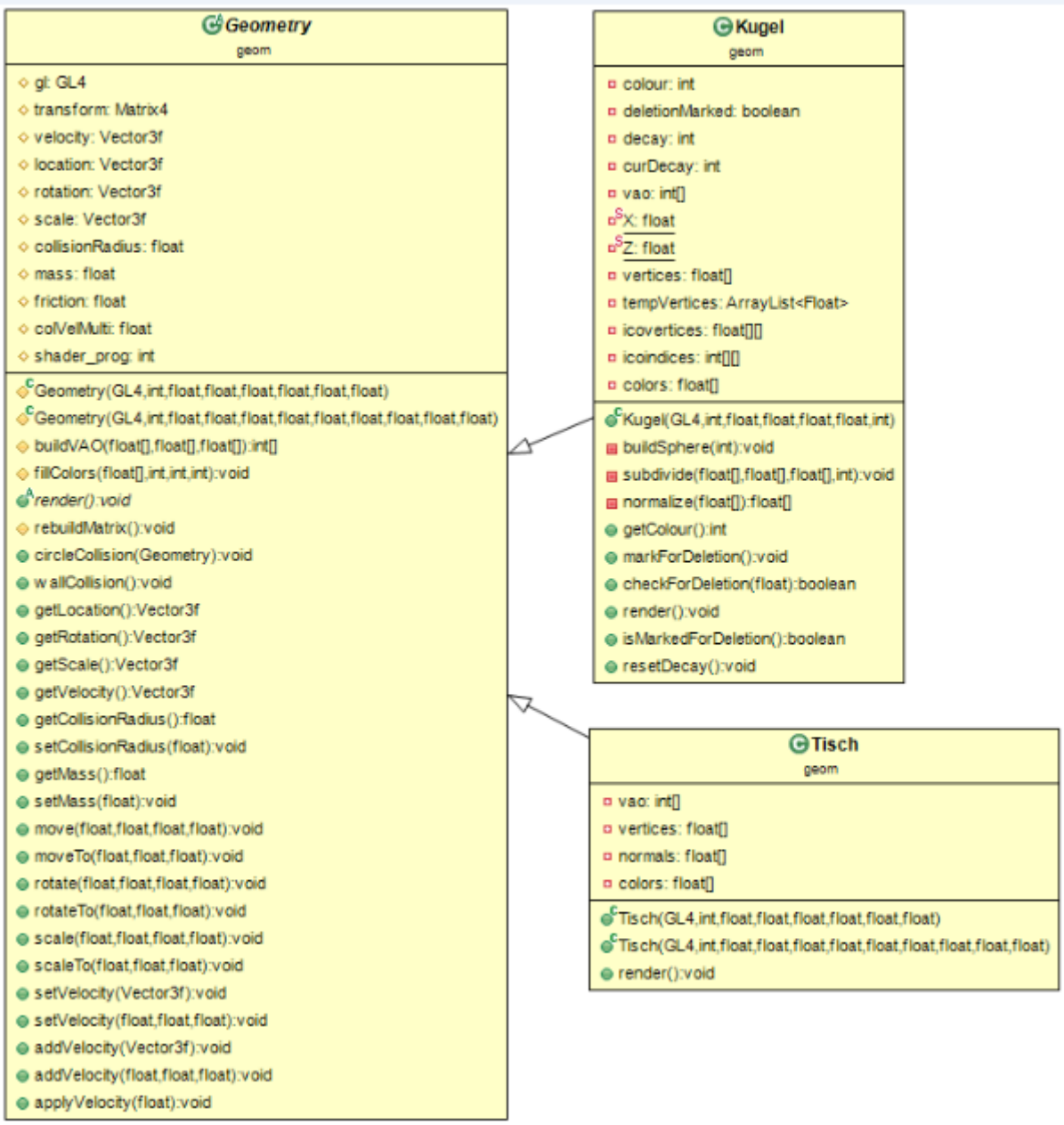
Der Algorithmus beginnt, wenn der Spieler die Kugel anstößt.
Wenn nun die Kugel das Loch trifft wird erst einmal überprüft, ob es sich hierbei um eine weiße Kugel handelt. Wenn es die weiße Kugel ist, dann ist die Runde für den aktuellen Spieler vorbei und der andere Spieler ist an der Reihe. Sollte es sich aber nicht um die Weiße sondern um eine andere Kugel handeln, wird abgefragt ob es sich um die schwarze Kugel handelt. Wenn dies der Fall ist wird nun überprüft ob der Spieler bereits jede seiner Kugeln eingelocht hat. Wenn es so ist gewinnt der Spieler die Runde und das Spiel ist vorbei. Sollten sich aber noch Kugeln der Farbe des Spielers auf dem Feld befinden, hat der Spieler verloren. Wenn die Kugeln aber weder Weiß noch Schwarz ist wird nun überprüft ob es sich um die Farbe des aktuellen Spielers handelt ist dies nicht der Fall, wird der Zähler der anderen Farbe erhöht und der andere Spieler ist an der Reihe. Hat die Kugel allerdings die Farbe des Spielers wird jener Zähler erhöht und der Spieler hat die Möglichkeit ein weiteres Mal an der Reihe zu sein.

Klassendiagramm




MatrixManager
managers
instance: MatrixManager
gl: GL4
viewMat: float[]
projectionMat: float[]
shm: ShaderManager
getInstance():MatrixManager
MatrixManager()
cameraLookAt(float[],float[],float[]):void
buildProjectionMatrix(float,float,float,float):void
updateUniforms():void
setGL(GL4):void
vec2float(Vector3f):float[]
StateManager
managers
instance: StateManager
stateList: HashMap<Integer,State>
activeState: State
getInstance():StateManager
StateManager()
addState(State):void
removeState(int):void
getActiveState():State
setActiveState(int):void
initStates(GLAutoDrawable):void
TouchDetector
managers
instance: TouchDetector
im: ImageManager
cm: CalibrationManager
touchPoints: ArrayList<TouchPoint>
getInstance():TouchDetector
TouchDetector()
getTouchPoints():ArrayList<TouchPoint>
updateTouchPoints():void
ShaderManager
managers
instance: ShaderManager
gl: GL4
SHADER_DIFFUSE: int
vertex_shader1: String[]
fragment_shader1: String[]
getInstance():ShaderManager
ShaderManager()
setGL(GL4):void
buildShaders():void
CalibrationManager
managers
instance: CalibrationManager
camPointsMat: Mat
screenPointsMat: Mat
camToScreenMat: Mat
resX: int
resY: int
getInstance():CalibrationManager
CalibrationManager()
setCalibrationPoints(ArrayList<Point>,ArrayList<Point>):void
translateToScreenCoords(ArrayList<TouchPoint>):ArrayList<TouchPoint>
calculateTransformMatrix():void
setResolution(int,int):void
getScreenWidth():int
getScreenHeight():int
ImageManager
managers
instance: ImageManager
webcam: VideoCapture
bgImage: Mat
image: Mat
frameCount: int
getInstance():ImageManager
ImageManager()
getImage():Mat
getBackgroundImage():Mat
captureFrame():boolean
calculateTouchPoints(int,int):ArrayList<TouchPoint>
cvtGrayscale():void
amplify():void
boxFilter(int):void
removeBackground():void
recaptureBackground():void
binaryThreshold(int):void
threshold(int):void


TouchPoint
utils
size: int
TouchPoint(Point,int)
TouchPoint(int,int,int)
getSize():int
Farbe
utils
WEISS: int
SCHWARZ: int
ROT: int
BLAU: int
Farbe()


Geometry
geom
gl: GL4
transform: Matrix4
velocity: Vector3f
location: Vector3f
rotation: Vector3f
scale: Vector3f
collisionRadius: float
mass: float
friction: float
colVelMulti: float
shader_prog: int
Geometry(GL4,int,float,float,float,float,float,float)
Geometry(GL4,int,float,float,float,float,float,float,float,float,float,float)
buildVAO(float[],float[],float[]):int[]
fillColors(float[],int,int,int):void
render():void
rebuildMatrix():void
circleCollision(Geometry):void
wallCollision():void
getLocation():Vector3f
getRotation():Vector3f
getScale():Vector3f
getVelocity():Vector3f
getCollisionRadius():float
setCollisionRadius(float):void
getMass():float
setMass(float):void
move(float,float,float,float):void
moveTo(float,float,float):void
rotate(float,float,float,float):void
rotateTo(float,float,float):void
scale(float,float,float,float):void
scaleTo(float,float,float):void
setVelocity(Vector3f):void
setVelocity(float,float,float):void
addVelocity(Vector3f):void
addVelocity(float,float,float):void
applyVelocity(float):void
Kugel
geom
colour: int
deletionMarked: boolean
decay: int
curDecay: int
vao: int[]
X: float
Z: float
vertices: float[]
tempVertices: ArrayList<Float>
icovertices: float[][]
icoindices: int[][]
colors: float[]
Kugel(GL4,int,float,float,float,float,int)
buildSphere(int):void
subdivide(float[],float[],float[],int):void
normalize(float[]):float[]
getColour():int
markForDeletion():void
checkForDeletion(float):boolean
render():void
isMarkedForDeletion():boolean
resetDecay():void
Tisch
geom
vao: int[]
vertices: float[]
normals: float[]
colors: float[]
Tisch(GL4,int,float,float,float,float,float,float)
Tisch(GL4,int,float,float,float,float,float,float,float,float,float,float)
render():void

- # Referenzen

http://antongerdelan.net/opengl/#onlinetuts Basistutorials für OpenGL

http://www.gamasutra.com/view/feature/3015/pool_hall_lessons_fast_accurate_.php?page=3 Tutorial für Kugelkollisionen

http://www.scriptscoop.net/t/17e0f9037cf1/java-2d-elastic-collisions-sticking-behaviour.html Lösung eines speziellen Problems bei den Kugelkollisionen

- # Anhang A - Quellcode

Klasse – main

```
package main;

import java.awt.Frame;
import java.awt.GraphicsEnvironment;
import java.awt.Rectangle;

import org.opencv.core.Core;

import com.jogamp.opengl.GL4;
import com.jogamp.opengl.GLAutoDrawable;
import com.jogamp.opengl.GLCapabilities;
import com.jogamp.opengl.GLEventListener;
import com.jogamp.opengl.GLProfile;
import com.jogamp.opengl.awt.GLCanvas;
import com.jogamp.opengl.util.Animator;
import com.jogamp.opengl.util.FPSAnimator;

import managers.CalibrationManager;
import managers.MatrixManager;
import managers.ShaderManager;
import managers.StateManager;
import managers.TouchDetector;
import states.BilliardState;
import states.CalibrationState;
import utils.State;

public class Main implements GLEventListener {

	static {
		// Load the native OpenCV library
		System.loadLibrary( Core.NATIVE_LIBRARY_NAME );
	}


	private long lastMillis;
	private TouchDetector td;
	private StateManager stm;
	private ShaderManager shm;
	private MatrixManager mm;

	/**
	 * Erstellt das Mainobjekt, die Managerobjekte
	 * und die benötigten States. Setzt außerdem den ersten
	 * aktiven State.
```

```
	 */
	public Main() {
		td = TouchDetector.getInstance();
		stm = StateManager.getInstance();
		shm = ShaderManager.getInstance();
		mm = MatrixManager.getInstance();

		//Hier wäre der CalibrationState gestartet worden, welcher dann von
sich aus später
		//auf den BilliardState gewechselt hätte.

		//stm.addState(new CalibrationState());
		stm.addState(new BilliardState());
		//stm.setActiveState(0);
		stm.setActiveState(1);

	}

	/**
	 * Main Methode
	 * Erstellt den Frame und grundlegende OpenGl Objekte
	 * @param args Argumente
	 */
	public static void main(String[] args) {
		GLProfile glp = GLProfile.getMaximum(true);
	  GLCapabilities caps = new GLCapabilities(glp);
	  GLCanvas canvas = new GLCanvas(caps);
	  canvas.addGLEventListener(new Main());

	  Animator an = new Animator(canvas);
	  an.start();

	  Frame frame = new Frame("MultiTouchDemo");
	   Rectangle b = GraphicsEnvironment.getLocalGraphicsEnvironment()

	.getDefaultScreenDevice().getDefaultConfiguration().getBounds();

	  frame.add(canvas);
	  frame.setSize((int)b.getWidth(), (int)b.getHeight());
	  frame.setUndecorated(true);
	  frame.requestFocus();
	  frame.setVisible(true);
	}

	/**
	 * Hauptprogrammzyklus
	 * Holt sich das aktuelle Kamerabild, wertet es aus
	 * und führt die update- und Rendermethoden des aktiven States aus.
	 * Berechnet außerdem die Framezeit.
	 */
	@Override
	public void display(GLAutoDrawable drawable) {
		long nanoTime = System.nanoTime();
		td.updateTouchPoints();
		State state = stm.getActiveState();
		if(state != null) {
			state.update(lastMillis);
			state.render(drawable);
		}
		lastMillis = (System.nanoTime() - nanoTime)/1000000;
	}
```

```
	@Override
	public void dispose(GLAutoDrawable drawable) {}

	/**
	 * Initialisiert die States und aktiviert grundlegende OpenGL Optionen
	 */
	@Override
	public void init(GLAutoDrawable drawable) {
		GL4 gl = drawable.getGL().getGL4();
		gl.glEnable(GL4.GL_DEPTH_TEST);

		CalibrationManager.getInstance().setResolution(1280, 1024);

		shm.setGL(gl);
		shm.buildShaders();
		mm.setGL(gl);
		stm.initStates(drawable);
	}

	@Override
	public void reshape(GLAutoDrawable drawable, int x, int y, int width, int
height) {
	}
}
```

Klasse – state

```
package utils;

import com.jogamp.opengl.GLAutoDrawable;

public interface State {
	/**
	 * Initmethode des States, wird anfangs einmalig aufgerufen
	 * @param drawable Drawable Referenz
	 */
	public void init(GLAutoDrawable drawable);

	/**
	 * Updatezyklus des States, wird von der Main Klasse kontinuierlich
aufgerufen
	 * @param lastMillis Zeit des letzten Frames
	 */
	public void update(float lastMillis);

	/**
	 * Renderzyklus des States, wird nach der Updatemethode ebenfalls
kontinuierlich aufgerufen
	 * @param drawable Drawable Referenz
	 */
	public void render(GLAutoDrawable drawable);

	/**
	 * Gibt die ID des States zurück
	 * @return ID des States
	 */
	public Integer getId();
}
```

Klasse – BillardState

```
package states;

import java.util.ArrayList;
import javax.vecmath.Vector3f;
import org.opencv.core.Point;
import com.jogamp.opengl.GL4;
import com.jogamp.opengl.GLAutoDrawable;
import geom.Kugel;
import geom.Tisch;
import managers.CalibrationManager;
import managers.MatrixManager;
import managers.ShaderManager;
import managers.StateManager;
import managers.TouchDetector;
import utils.Farbe;
import utils.State;
import utils.TouchPoint;

public class BilliardState implements State {

	private GL4 gl;

	private TouchDetector td = TouchDetector.getInstance();
	private ShaderManager shm = ShaderManager.getInstance();
	private MatrixManager mm = MatrixManager.getInstance();
	private CalibrationManager cm = CalibrationManager.getInstance();

	private ArrayList<TouchPoint> touchPoints = new ArrayList<TouchPoint>();

    private Tisch tisch;
    private Kugel weiss;
    private ArrayList<Kugel> kugeln = new ArrayList<Kugel>();

    private TouchPoint lastPoint;

    //Kugelgroesse (Radius)
    private float size = 0.03f;
    //Distanz zwischen Kugeln beim Start
    private float dist = 0.06f;

    //Minimale und maximaler Unterschied der Punkte zweier Frames,
    //der zum Schuss fuehrt
    private float minShotDelta = 15, maxShotDelta = 30;

    //Schuss Multiplier
    private float shotStrengthMuli = 2.5f;

    //Kamera Optionen
    private boolean cameraFixed;
    private int cameraUnfixDelay = 100, curUnfixDelay = 0;
    private Point camIdlePoint = new Point(640,512);

    private float  camMultiX = 0.05f,
                   camMultiY = 0.05f;

    private float  camAngleX = 0.5f * (float)Math.PI,
                   camAngleY =  0.75f * (float)Math.PI,
```

```
                camRadius = 1f;


        /* Spiel Variablen */

        //Kugel Anzahlen
        private int kugelnRot = 7, kugelnBlau = 7;

        //Aktueller Spieler
        private int aktSpieler = Farbe.ROT;
        private boolean naechsterSpieler = true;

        //Weisse Kugel außerhalb
        private boolean weissAus;

        //Falsche Kugel versenkt?
        private boolean rBlaueVersenkt;
        private boolean bRoteVersenkt;

        //In dieser Runde schon gestoßen?
        private boolean shot = false;

        //Spiel zu Ende?
        private boolean gameEnd;


        /**
         * Initialisiert die Geometrieobjekte,
         * die Kamera und das Licht
         */
        @Override
        public void init(GLAutoDrawable drawable) {
              gl = drawable.getGL().getGL4();

              //Geometrie Objekte
              tisch = new Tisch(gl, shm.SHADER_DIFFUSE,
                          0,0,0,              //Ort
                          1,1,1               //Groesse
              );
                 //                      GL  Shader
        Ort                                          Groesse  Farbe
                 weiss = new Kugel(gl, shm.SHADER_DIFFUSE,       0.0f, size, 1f,
                                 size, Farbe.WEISS);

              kugeln.add(new Kugel(gl, shm.SHADER_DIFFUSE,       0.0f, size, 0f,
                                 size, Farbe.ROT));

              kugeln.add(new Kugel(gl, shm.SHADER_DIFFUSE,       -(dist/2), size,
0f-dist,           size, Farbe.ROT));
              kugeln.add(new Kugel(gl, shm.SHADER_DIFFUSE,       (dist/2), size,
0f-dist,           size, Farbe.BLAU));

              kugeln.add(new Kugel(gl, shm.SHADER_DIFFUSE,       -dist, size, 0f-
2*dist,                  size, Farbe.BLAU));
              kugeln.add(new Kugel(gl, shm.SHADER_DIFFUSE,       0.0f, size, 0f-
2*dist,                  size, Farbe.SCHWARZ));
              kugeln.add(new Kugel(gl, shm.SHADER_DIFFUSE,       dist, size, 0f-
2*dist,                  size, Farbe.ROT));

              kugeln.add(new Kugel(gl, shm.SHADER_DIFFUSE,       -(3*dist/2), size,
0f-3*dist,   size, Farbe.BLAU));
```

```
            kugeln.add(new Kugel(gl, shm.SHADER_DIFFUSE,        -(dist/2), size,
0f-3*dist,          size, Farbe.ROT));
            kugeln.add(new Kugel(gl, shm.SHADER_DIFFUSE,        (dist/2), size,
0f-3*dist,          size, Farbe.BLAU));
            kugeln.add(new Kugel(gl, shm.SHADER_DIFFUSE,        (3*dist/2), size,
0f-3*dist,    size, Farbe.BLAU));

            kugeln.add(new Kugel(gl, shm.SHADER_DIFFUSE,        -2*dist, size, 0f-
4*dist,             size, Farbe.BLAU));
            kugeln.add(new Kugel(gl, shm.SHADER_DIFFUSE,        -dist, size, 0f-
4*dist,                   size, Farbe.ROT));
            kugeln.add(new Kugel(gl, shm.SHADER_DIFFUSE,        0.0f, size, 0f-
4*dist,                   size, Farbe.BLAU));
            kugeln.add(new Kugel(gl, shm.SHADER_DIFFUSE,        dist, size, 0f-
4*dist,                   size, Farbe.ROT));
            kugeln.add(new Kugel(gl, shm.SHADER_DIFFUSE,        2*dist, size, 0f-
4*dist,             size, Farbe.ROT));

            // Projektionsmatrix erstellen
            float aspect =  16 / 9;
            mm.buildProjectionMatrix((float)Math.toRadians(60), aspect, 0.1f,
100);

            // Lichtinformationen an Shader uebergeben
            float[] lightPosition = {0,3,0};
            float[] lightIntensity = {1f};

            int lightPosLoc = gl.glGetUniformLocation(shm.SHADER_DIFFUSE,
"lightPos");
            gl.glUniform3fv(lightPosLoc, 1, lightPosition, 0);
            int lightIntLoc = gl.glGetUniformLocation(shm.SHADER_DIFFUSE,
"lightInt");
            gl.glUniform1fv(lightIntLoc, 1, lightIntensity, 0);

//          mm.cameraLookAt(
//                      new float[]{0,3,0},
//                      new float[]{0,0,0},
//                      new float[]{1,0,0}
//          );

            updateCamera(true);
      }



      /**
       * Haupt Update-Zyklus des Billard Spiels
       */
      @Override
      public void update(float lastMillis) {
            float delta = lastMillis/1000;

            if(!gameEnd) {
                  //Neueste Touchinformationen holen und alten Punkt speichern
                  if(touchPoints.size() > 0)
                        lastPoint = touchPoints.get(0);

                  touchPoints = td.getTouchPoints();

                  //Ueberprüfe Kamera Rotation
                  if(!cameraFixed) {
```

```
                        if(touchPoints.size() > 0) {
                                TouchPoint tp = touchPoints.get(0);
                                int dx = (int) (tp.x - camIdlePoint.x);
                                //int dy = (int) (tp.y - camIdlePoint.y);

                                float nX = (float)dx / (float)cm.getScreenWidth();
                                //float nY = (float)dy /
(float)cm.getScreenHeight();

                                camAngleX += nX * camMultiX;
                                //camAngleY += nY * camMultiY;

                                //Kamera updaten (Folge Weiß, wenn nicht
geschossen wurde)
                                updateCamera(!shot);
                        }
                } else {
                        //Nach Schuss für eine kurze Zeit keine Kameradrehung
erlauben
                        curUnfixDelay += lastMillis;
                        if(curUnfixDelay >= cameraUnfixDelay) {
                                cameraFixed = false;
                                curUnfixDelay = 0;
                        }
                }


                //Auf nächsten Spieler wechseln
                //wenn sich die Kugeln (fast) nicht mehr bewegen
                if(shot) {
                        boolean nextCheck = false;
                        //Wenn sich die weisse Kugel fast nicht mehr bewegt oder
gerade aus dem Feld fällt (seit mindestens dem decay der Kugel)
                        if((weiss.getVelocity().length() < 0.01f ||
weiss.checkForDeletion(lastMillis))) {
                                nextCheck = true;

                                //Ueberpruefe, ob alle Kugeln sich ebenfalls nicht
mehr bewegen
                                for(Kugel k : kugeln){
                                        if(k.getVelocity().length() > 0.01f) {
                                                nextCheck = false;
                                                break;
                                        }
                                }
                        }
                        if(nextCheck) {
                                nextCheck = false;

                                //Kugeln zum Halten bringen
                                weiss.setVelocity(0,0,0);
                                for(Kugel k : kugeln){
                                        k.setVelocity(0,0,0);
                                }

                                //Falls nötig, die weisse Kugel wieder an den
Startpunkt legen
                                if(weissAus) {
                                        weiss = new
Kugel(gl,shm.SHADER_DIFFUSE,0.0f, size, 1f,size,Farbe.WEISS);
                                }
```

```
                                //Kamera Updaten
                                updateCamera(true);

                                System.out.println("================");

                                //Naechsten Spieler setzen, falls nötig
                                if(naechsterSpieler || weissAus) {
                                        weissAus = false;

                                        if(aktSpieler == Farbe.ROT) {
                                                aktSpieler = Farbe.BLAU;
                                                System.out.println("Nächster Spieler
ist Blau");
                                        } else {
                                                aktSpieler = Farbe.ROT;
                                                System.out.println("Nächster Spieler
ist Rot");
                                        }
                                }

                                //Wenn in einer Runde etwas getroffen und
daraufhin nichts mehr getroffen wurde,
                                //soll der Spieler wieder gewechselt werden.
                                naechsterSpieler = true;

                                //Schuss beendet
                                shot = false;
                                lastPoint = null;

                                rBlaueVersenkt = false;
                                bRoteVersenkt = false;

                        }
                } else {
                        //Wenn noch nicht geschossen wurde, pruefe auf schnelle
Wischgeste nach oben
                        if(touchPoints.size() > 0 && lastPoint != null) {
                                int dy = (int) (touchPoints.get(0).y -
lastPoint.y);

                                if(dy > minShotDelta) {
                                        //System.out.println("dy:"+dy);
                                        //Stärke des Schusses berechnen
                                        float strength = Math.min(((dy -
minShotDelta) / (maxShotDelta - minShotDelta)), 1);
                                        //System.out.println("strength:"+strength);

                                        //Vektor in Kamera Richtung mit Länge
strength*multiplier finden
                                        float x =
strength*(float)Math.cos(camAngleX);
                                        float z =
strength*(float)Math.sin(camAngleX);

                                        Vector3f shotVector = new Vector3f(x, 0,
z);
                                        shotVector.scale(shotStrengthMuli);
                                        shotVector.negate();
                                        weiss.setVelocity(shotVector.x, 0,
shotVector.z);
```

```
      //System.out.println("strength:"+strength+", vector
length:"+shotVector.length());
                                    shot = true;

                                    cameraFixed = true;
                              }
                        } else {
                              lastPoint = null;
                        }
                  }
            }

            //Pruefe auf Kollision aller farbigen Kugeln + Schwarz mit anderen
Kugeln/Waenden
            //und ueberpruefe ob sie geloescht werden muessen
            for (int i = 0; i < kugeln.size(); i++) {

                  //Kollisionschecks
                for (int k = i+1; k < kugeln.size(); k++) {
                    kugeln.get(i).circleCollision(kugeln.get(k));
                }
                kugeln.get(i).wallCollision();

                //Pruefe ob sich Kugel i im Feld befindet, wenn nicht, merke sie
zum loeschen vor
                if(!isBallInField(kugeln.get(i)) &&
!kugeln.get(i).isMarkedForDeletion()) {

                  //Farbe der herausgerollten Kugel prüfen
                  if(kugeln.get(i).getColour() == Farbe.BLAU) {
                        //Blau
                        kugelnBlau--;
                        System.out.println("-> Spieler versenkte eine Blaue
Kugel");
                        if(aktSpieler == Farbe.BLAU && !bRoteVersenkt) {
                              naechsterSpieler = false;
                        } else {
                              rBlaueVersenkt = true;
                        }
                  } else if(kugeln.get(i).getColour() == Farbe.ROT) {
                        //Rot
                        kugelnRot--;
                        System.out.println("-> Spieler versenkte eine Rote
Kugel");
                        if(aktSpieler == Farbe.ROT && !rBlaueVersenkt) {
                              naechsterSpieler = false;
                        } else {
                              bRoteVersenkt = true;
                        }

                  } else {
                        //Schwarz
                        if(aktSpieler == Farbe.ROT) {
                              if(kugelnRot <= 0) {
                                    endGame(Farbe.ROT);
                              } else endGame(Farbe.BLAU);
                        } else
                              if(kugelnBlau <= 0) {
                                    endGame(Farbe.BLAU);
                              } else endGame(Farbe.ROT);
```

```
                }

                //Zur baldigen Loeschung vormerken und fallen lassen
                kugeln.get(i).markForDeletion();
                kugeln.get(i).setVelocity(0, -1, 0);
            }

            //Wenn die Zeit (decay) der Kugel i um ist, loesche sie aus dem
Spiel
            if(kugeln.get(i).checkForDeletion(lastMillis)) {
                kugeln.remove(i);
            }
        }

        //Ueberpruefe, ob sich die weisse Kugel im Feld befindet
        if(!isBallInField(weiss)) {
            if(!weiss.isMarkedForDeletion()) {
                //Wenn nein, dann boolean setzen und Kugel fallen lassen
                weiss.setVelocity(0, -1, 0);
                weiss.markForDeletion();
                weissAus = true;
                System.out.println("-> Spieler versenkte die weiße
Kugel!");
            }
        }

        //Weisse Kugel mit allen anderen + Waenden kollidieren lassen
        for(Kugel kugel : kugeln) {
            weiss.circleCollision(kugel);
        }
        weiss.wallCollision();

        //Die Frame-skalierte Geschwindigkeit/Richtung und Reibungswiderstand
jeder Kugel anwenden
        weiss.applyVelocity(delta);
        for(Kugel kugel : kugeln) {
            kugel.applyVelocity(delta);
        }
    }


    /**
     * Renderzyklus des Billardspiels
     * Hier werden alle Objekte gerendert
     */
    @Override
    public void render(GLAutoDrawable drawable) {
        gl.glClear(GL4.GL_COLOR_BUFFER_BIT | GL4.GL_DEPTH_BUFFER_BIT);
        gl.glClearColor(0.2f, 0.2f, 0.2f, 1.0f);

        //Tisch und alle Kugeln rendern
        tisch.render();
        weiss.render();
        for(Kugel k : kugeln) {
            k.render();
        }

    }

    /**
     * Überprüft, ob die gegebene Kugel sich auf dem Spielfeld befindet
```

```
	 * @param k Die zu überprüfende Kugel
	 * @return Boolean, ob sich die Kugel auf dem Feld befindet
	 */
	public boolean isBallInField(Kugel k) {
		if(Math.abs(k.getLocation().x) >= 0.635f + size/2 ||
Math.abs(k.getLocation().z) >= 1.27f + size/2) {
			return false;
		}
		return true;
	}

	/**
	 * Löscht die Kugel aus der Liste und damit aus dem Spiel
	 * @param k Die zu löschende Kugel
	 */
	public void removeBall(Kugel k) {
		if(kugeln.contains(k)) {
			kugeln.remove(k);
		}
	}

	//Kamera in einer Bounding Sphere mit r = camRadius um die weiße Kugel
platzieren und mit camAngleX/Y ausrichten
	/**
	 * Richtet die Kamera aus, entweder am Ursprung oder an der weißen Kugel.
	 * Dabei werden die aktuellen Rotationsinformationen verwendet
	 * @param followWhite Boolean, ob der weißen Kugel gefolgt werden soll
	 */
	private void updateCamera(boolean followWhite) {
		Vector3f loc;
		if(followWhite) {
			loc = weiss.getLocation();
		} else {
			loc = new Vector3f(0,0,0);
		}

		mm.cameraLookAt(
				new
float[]{loc.x+camRadius*(float)Math.cos(camAngleX),camRadius*(float)Math.sin(camAn
gleY),loc.z+camRadius*(float)Math.sin(camAngleX)},
				mm.vec2float(loc),
				new float[]{0,1,0}
		);
	}

	/**
	 * Beendet das Spiel mit dem angegebenen Gewinner
	 * @param winner Farbkonstante, welche den gewinnenden Spieler symbolisiert
	 */
	private void endGame(int winner) {
		System.out.println("Spieler "+(winner == Farbe.ROT?"ROT":"BLAU")+"
hat gewonnen!");
		gameEnd = true;
	}

	@Override
	public Integer getId() {
		return 1;
	}
}
```

Klasse – CalibrationState

```
package states;

import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.geom.Ellipse2D;
import java.util.ArrayList;

import org.opencv.core.Point;

import com.jogamp.opengl.GL4;
import com.jogamp.opengl.GLAutoDrawable;
import com.jogamp.opengl.util.awt.Overlay;

import managers.CalibrationManager;
import managers.MatrixManager;
import managers.StateManager;
import managers.TouchDetector;
import utils.State;
import utils.TouchPoint;

public class CalibrationState implements State {

	/* War ein Versuch, aber das Overlay funktioniert aufgrund einer
	 * GLException nicht. Aber das Billardspiel funktioniert auch ohne eine exakte Kalibrierung,
	 * die wäre nur bei z.B. einem Malprogramm von Nöten gewesen.
	 */

	GL4 gl;

	Overlay overlay;
	Graphics2D oGraphics;
	CalibCircle[] calibCircles = new CalibCircle[4];

	TouchDetector td;
	StateManager sm;
	MatrixManager mm;
	CalibrationManager cm;

	ArrayList<TouchPoint> touchPoints;
	TouchPoint[] camPoints = new TouchPoint[4];

	float delay = 1000;

	/**
	 * Initialisiert die Grafiken zur Kalibrierung
	 * sowie das AWT Overlay
	 */
	@Override
	public void init(GLAutoDrawable drawable) {
		gl = drawable.getGL().getGL4();
		td = TouchDetector.getInstance();
		sm = StateManager.getInstance();
		mm = MatrixManager.getInstance();
		cm = CalibrationManager.getInstance();
```

```
		//Erstellt ein Overlay über dem Canvas um 2D Zeichnung zu
vereinfachen
		overlay = new Overlay(drawable);
		oGraphics = overlay.createGraphics();

		System.out.println("blubb");

		//Erstelle Kreise
		calibCircles[0] = new CalibCircle(50, 50, 50, 50);
		calibCircles[1] = new CalibCircle(cm.getScreenWidth()-50, 50, 50,
50);
		calibCircles[2] = new CalibCircle(50, cm.getScreenHeight()-50, 50,
50);
		calibCircles[3] = new CalibCircle(cm.getScreenWidth()-50,
cm.getScreenHeight()-50, 50, 50);
	}

	/**
	 * Updatezyklus des Kalibrierungsstates
	 */
	@Override
	public void update(float lastMillis) {
		touchPoints = td.getTouchPoints();

		// Wenn Touchpunkte gefunden wurden,
		// dann rufe die Update Methode der Kreise auf
		if(touchPoints.size() > 0) {
			TouchPoint tp = touchPoints.get(0);

			int i = 0;
			for(CalibCircle circle : calibCircles) {
				circle.update(lastMillis, tp);
				if(circle.isCalibrated())
					camPoints[i] = tp;
				i++;
			}
		}

		//Überprüfe ob alle Kreise kalibriert wurden
		boolean completeCheck = true;
		for(int i = 0; i < 4; i++) {
			if(camPoints[i] == null) {
				completeCheck = false;
				break;
			}
		}

		//Wenn alle kalibriert sind, dann stelle die Punktepaarlisten für den
		//Calibration Manager auf, übergebe diese und wechsel auf den Billard
State
		if(completeCheck) {
			ArrayList<Point> camPointList = new ArrayList<Point>();
			ArrayList<Point> screenPointList = new ArrayList<Point>();

			for(int i = 0; i < 4; i++) {
				camPointList.add(camPoints[i]);
				screenPointList.add(new
Point(calibCircles[i].getX(),calibCircles[i].getY()));
			}
			cm.setCalibrationPoints(camPointList, screenPointList);
			sm.setActiveState(1);
```

```
		}
	}

	/**
	 * Rendere alle Kalibrierungskreise
	 */
	@Override
	public void render(GLAutoDrawable drawable) {
		for(CalibCircle circle : calibCircles) {
			//oGraphics.setColor(circle.getColor());
			//System.out.println(oGraphics);
			//oGraphics.draw(circle);
		}
	}

	@Override
	public Integer getId() {
		return 0;
	}



	private class CalibCircle extends Ellipse2D.Float {
		private float currentTime = 0;
		private boolean calibrated;

		/**
		 * Erstellt einen Kalibrierungskreis
		 * @param x X-Position des Kreises
		 * @param y Y-Position des Kreises
		 * @param w Breite des Kreises
		 * @param h Höhe des Kreises
		 */
		public CalibCircle(int x, int y, int w, int h) {
			super(x,y,w,h);
		}

		/**
		 * Updatezyklus des Kreises,
		 * verrechnet die Framezeit und überprüft den Fortschritt der
		 * Kalibrierung
		 * @param delta Framezeit
		 * @param tp Übergabe des Touchpunktes
		 */
		public void update(float delta, TouchPoint tp) {
			if(currentTime < delay && !calibrated) {
				if(contains(tp.x, tp.y)) {
					currentTime += delta;
				} else {
					currentTime = 0;
				}
			} else {
				calibrated = true;

			}
		}

		/**
		 * Gibt eine Farbe zurück, welche den Fortschritt wiederspiegelt
		 * @return Farbe
		 */
```

```
		public Color getColor() {
			int r = 0;
			int g = Math.min((int) ((currentTime/delay)*255),255);
			int b = Math.min((int) (1-((currentTime/delay)*255)),255);
			return new Color(r,g,b);
		}

		/**
		 * Gibt zurück, ob der Kreis kalibriert ist
		 * @return Boolean ob der Kreis kalibriert ist
		 */
		public boolean isCalibrated() {
			return calibrated;
		}
	}

}
```

Klasse – Matrix Manager

```
package managers;

import javax.vecmath.Vector3f;

import com.jogamp.opengl.GL4;
import com.jogamp.opengl.math.FloatUtil;
import com.jogamp.opengl.math.Matrix4;

public class MatrixManager {
	private static MatrixManager instance;
	public static MatrixManager getInstance() {
		if (instance == null) {
			instance = new MatrixManager();
		}
		return instance;
	}

	/**
	 * Erstellt einen Matrix Manager und initialiesiert die Matrizen
	 */
	private MatrixManager() {
		viewMat = FloatUtil.makeIdentity(new float[16]);
		projectionMat = FloatUtil.makeIdentity(new float[16]);
	};

	private GL4 gl;
	private float[] viewMat, projectionMat;
	private ShaderManager shm = ShaderManager.getInstance();

	/**
	 * Erstellt eine neue View Matrix und aktualisiert die Uniform Variablen
	 * @param eye Vektor für den Augenpunkt
	 * @param center Vektor des anzuschauenden Punktes
	 * @param up Up-Vektor
	 */
	public void cameraLookAt(float[] eye, float[] center, float[] up) {
		viewMat = FloatUtil.makeLookAt(viewMat, 0, eye, 0, center, 0, up, 0,
new Matrix4().getMatrix());
		updateUniforms();
	}

	/**
	 * Erstellt eine neue perspektivische Projektionsmatrix und aktualisiert die
Uniform Variablen
	 * @param fieldOfView Das zu verwendende Field of View (in Radialwerten)
	 * @param aspectRatio Das Seitenverhältnis des Bildschirms
	 * @param nearPlane Die Entfernung der Near-Plane
	 * @param farPlane Die Entfernung der Far-Plane
	 */
	public void buildProjectionMatrix(float fieldOfView, float aspectRatio,
float nearPlane, float farPlane) {
		projectionMat = FloatUtil.makePerspective(projectionMat, 0, true,
fieldOfView, aspectRatio, nearPlane, farPlane);
		updateUniforms();
	}

	/**
```

```
	 * Lädt die aktuellen Matrizen als Uniform Variablen in den VRAM der
Grafikkarte
	 */
	private void updateUniforms() {
		gl.glUseProgram(shm.SHADER_DIFFUSE);
		gl.glUniformMatrix4fv(gl.glGetUniformLocation(shm.SHADER_DIFFUSE,
"view"), 1, false, viewMat,0);
		gl.glUniformMatrix4fv(gl.glGetUniformLocation(shm.SHADER_DIFFUSE,
"projection"), 1, false, projectionMat,0);
	}

	/**
	 * Setzt die GL4 Referenz
	 * @param gl Die zu setzende Referenz
	 */
	public void setGL(GL4 gl) {
		this.gl = gl;
	}

	/**
	 * Wandelt einen Vektor in ein Float-Array um
	 * @param v Der umzuwandelnde Vektor
	 * @return Das entstandene Float-Array
	 */
	public float[] vec2float(Vector3f v) {
		return new float[]{v.x,v.y,v.z};
	}
}
```

Klasse – StateManager

```
package managers;

import java.util.HashMap;
import com.jogamp.opengl.GLAutoDrawable;
import utils.State;

public class StateManager {
	private static StateManager instance;
	public static StateManager getInstance() {
		if (instance == null) {
			instance = new StateManager();
		}
		return instance;
	}

	/**
	 * Erstellt einen State Manager
	 */
	private StateManager() {};

	private HashMap<Integer, State> stateList = new HashMap<Integer, State>();
	private State activeState;
```

```
	/**
	 * Fügt einen State der State Liste hinzu
	 * @param state Der hinzuzufügende State
	 */
	public void addState(State state) {
		stateList.put(state.getId(), state);
	}

	/**
	 * Löscht einen State aus der State Liste
	 * @param id Die ID des zu löschenden States
	 */
	public void removeState(int id) {
		stateList.remove(id);
	}

	/**
	 * Gibt den aktuell als aktiv gesetzten State zurück
	 * @return Der aktive State
	 */
	public State getActiveState() {
		return activeState;
	}

	/**
	 * Setzt den aktiven State
	 * @param id ID des als Aktiv zu setzenden States
	 */
	public void setActiveState(int id) {
		activeState = stateList.get(id);
	}

	/**
	 * Ruft die Init-Methoden aller States auf
	 * @param drawable Drawable Referenz
	 */
	public void initStates(GLAutoDrawable drawable) {
		for(State state : stateList.values()) {
			state.init(drawable);
		}
	}
}
```

Klasse – CalibrationManager

```
package managers;

import java.util.ArrayList;
import java.util.HashMap;

import org.opencv.core.Core;
import org.opencv.core.Mat;
import org.opencv.core.Point;
import org.opencv.imgproc.Imgproc;
import org.opencv.utils.Converters;


import utils.TouchPoint;

public class CalibrationManager {
	private static CalibrationManager instance;
	public static CalibrationManager getInstance() {
		if(instance == null) {
			instance = new CalibrationManager();
		}
		return instance;
	}

	/**
	 * Erstellt einen Calibration Manager und fügt
	 * eine Standardkalibierung ein
	 */
	private CalibrationManager() {
		/* Test */
		ArrayList<Point> camPoints = new ArrayList<Point>();
		camPoints.add(new Point(0,0));
		camPoints.add(new Point(640,0));
		camPoints.add(new Point(0,480));
		camPoints.add(new Point(640,480));

		ArrayList<Point> screenPoints = new ArrayList<Point>();
		screenPoints.add(new Point(0,0));
		screenPoints.add(new Point(1280,0));
		screenPoints.add(new Point(0,1024));
		screenPoints.add(new Point(1280,1024));

		this.setCalibrationPoints(camPoints, screenPoints);
	}

	private Mat camPointsMat;
	private Mat screenPointsMat;
	private Mat camToScreenMat;

	private int resX, resY;

	/**
	 * Ersetzt die aktuellen Kalibierungspunkte durch die angegebenen.
	 * @param camPoints Eine 4-Eintrags Point-Arraylist der
Kamerakoordinatenpunkte
	 * @param screenPoints Eine 4-Eintrags Point-Arraylist der
Bildschirmkoordinatenpunkte
	 */
```

```
	public void setCalibrationPoints(ArrayList<Point> camPoints,
ArrayList<Point> screenPoints) {
		if(camPoints.size() >= 4 && screenPoints.size() >= 4) {
			camPointsMat = Converters.vector_Point2f_to_Mat(camPoints);
			screenPointsMat =
Converters.vector_Point2f_to_Mat(screenPoints);

			calculateTransformMatrix();
		}
	}

	/**
	 * Errechnet die Bildschirmkoordinaten jedes Punktes in der angegebenen
Liste
	 * @param touchPoints Die Liste der umzurechnenden Punkte
	 * @return Die aktualisierte Liste
	 */
	public ArrayList<TouchPoint> translateToScreenCoords(ArrayList<TouchPoint>
touchPoints) {
		/* Wenn es Touchpunkte gibt */
		if(touchPoints.size() > 0) {
			ArrayList<TouchPoint> newTouchPoints = new
ArrayList<TouchPoint>();

			/* Konvertiere Touchpunkte in Matrix */
			Mat touchPointsMat = Converters.vector_Point2f_to_Mat(new
ArrayList<Point>(touchPoints));
			Mat newTouchPointsMat = new Mat();

			/* Transformiere Touchpunkte in Bildschirmkoordinaten */
			Core.perspectiveTransform(touchPointsMat, newTouchPointsMat,
camToScreenMat);

			ArrayList<Point> newPoints = new ArrayList<Point>();

			/* Konvertiere Matrix zurück in Punkte */
			Converters.Mat_to_vector_Point2f(newTouchPointsMat, newPoints);

			/* Erstelle neue Touchpunkte an neuen Positionen mit alter
Größeninformation */
			for(int i = 0; i < newPoints.size(); i++) {
				newTouchPoints.add(new
TouchPoint(newPoints.get(i),touchPoints.get(i).getSize()));
			}
			//System.out.println(",Neu:
"+newTouchPoints.get(0).x+","+newTouchPoints.get(0).y);
			return newTouchPoints;
		}
		return touchPoints;
	}

	/**
	 * Berechnet die Transformationsmatrix zur Überführung in
Bildschirmkoordinaten neu
	 */
	private void calculateTransformMatrix() {
		camToScreenMat = Imgproc.getPerspectiveTransform(camPointsMat,
screenPointsMat);
	}

	/**
```

```
	 * Setzt die Bildschirmauflösung, diese Info wird von anderen Klassen
verwendet
	 * @param x Neue Auflösungsbreite
	 * @param y Neue Auflösungshöhe
	 */
	public void setResolution(int x, int y) {
		resX = x;
		resY = y;
	}

	/**
	 * Gibt die Breite der Auflösung zurück
	 * @return Die Breite der Auflösung
	 */
	public int getScreenWidth() {
		return resX;
	}

	/**
	 * Gibt die Höhe der Auflösung zurück
	 * @return Die Höhe der Auflösung
	 */
	public int getScreenHeight() {
		return resY;
	}
}
```

Klasse – TouchDetector

```
package managers;

import java.util.ArrayList;
import utils.TouchPoint;

public class TouchDetector {
	private static TouchDetector instance;
	public static TouchDetector getInstance() {
		if(instance == null) {
			instance = new TouchDetector();
		}
		return instance;
	}
	private TouchDetector() {
		touchPoints = new ArrayList<TouchPoint>();
		im = ImageManager.getInstance();
		cm = CalibrationManager.getInstance();
	};


	private ImageManager im;
	private CalibrationManager cm;
	private ArrayList<TouchPoint> touchPoints;

	/**
```

```
	 * Gibt die aktuelle Liste der Touchpunkte zurück
	 * @return Die aktuelle Touchpunkt-Liste
	 */
	public ArrayList<TouchPoint> getTouchPoints() {
		return touchPoints;
	}

	/**
	 * Führt die wesentlichen Schritte zur Erkennung und
	 * Berechnung der Touchpunkte in einer bestimmten Reihenfolge aus
	 * und speichert diese in der ArrayList
	 */
	public void updateTouchPoints() {
		if(!im.captureFrame())
			return;
		im.cvtGrayscale();
		im.removeBackground();
		im.boxFilter(10);
		im.threshold(12);
		im.amplify();
		im.binaryThreshold(40);
		touchPoints = im.calculateTouchPoints(50,3000);
		touchPoints = cm.translateToScreenCoords(touchPoints);
	}
}
```

Klasse – ImageManager

```
package managers;

import java.util.ArrayList;
import java.util.List;

import org.opencv.core.Core;
import org.opencv.core.Mat;
import org.opencv.core.MatOfPoint;
import org.opencv.core.Point;
import org.opencv.core.Rect;
import org.opencv.core.Scalar;
import org.opencv.core.Size;
import org.opencv.highgui.VideoCapture;
import org.opencv.imgproc.Imgproc;

import utils.TouchPoint;

public class ImageManager {
	private static ImageManager instance;
	public static ImageManager getInstance() {
		if (instance == null) {
			instance = new ImageManager();
		}
		return instance;
	}
```

```
	/**
	 * Erstellt einen Image Manager und initialisiert die Webcam
	 */
	private ImageManager() {
		webcam = new VideoCapture(0);
		webcam.set(5, 60);
	};

	private VideoCapture webcam;
	private Mat bgImage = new Mat(), image = new Mat();
	private int frameCount;

	/**
	 * Gibt das aktuelle Webcambild zurück
	 * @return Das aktuelle Webcambild
	 */
	public Mat getImage() {
		return image;
	}

	/**
	 * Gibt das aktuelle Hintergrundbild zurück
	 * @return Das aktuelle Hintergrundbild
	 */
	public Mat getBackgroundImage() {
		return bgImage;
	}

	/**
	 * Erfasst ein neues Bild
	 * @return Boolean, ob dies geglückt ist
	 */
	public boolean captureFrame() {
		frameCount++;
		return webcam.read(image);
	}

	/**
	 * Sucht nach Touchpunkten, erstellt eine Liste und gibt diese zurück
	 * @param minSize Minimale Größe der Touchpunkte
	 * @param maxSize Maximale Größe der Touchpunkte
	 * @return Die gefüllte ArrayList
	 */
	public ArrayList<TouchPoint> calculateTouchPoints(int minSize, int maxSize)
{
		ArrayList<TouchPoint> touchPoints = new ArrayList<TouchPoint>();
		List<MatOfPoint> contours = new ArrayList<MatOfPoint>();
		if(!bgImage.empty()) {
			Imgproc.findContours(image, contours, new Mat(),
Imgproc.RETR_LIST,Imgproc.CHAIN_APPROX_SIMPLE);
			int i = 0;
			for(MatOfPoint contour : contours) {
				Rect rect = Imgproc.boundingRect(contour);
				if(rect.area() >= minSize && rect.area() < maxSize) {
					i++;
					int x = rect.x + rect.width/2;
					int y = rect.y + rect.height/2;
					touchPoints.add(new TouchPoint(x, y,
rect.width*rect.height));
					Core.rectangle(image, new Point(rect.x,rect.y),
new Point(rect.x+rect.width,rect.y+rect.height), new Scalar(255,0,0));
```

```
                        }
                    }
                    //if(!touchPoints.isEmpty())

        //System.out.print(touchPoints.get(0).x+","+touchPoints.get(0).y);
            }
            return touchPoints;
        }

        /**
         * Wendet einen Grauwertfilter an
         */
        public void cvtGrayscale() {
            Imgproc.cvtColor(image, image, Imgproc.COLOR_RGB2GRAY);
        }

        /**
         * Wendet eine quadratische Verstärkung an
         */
        public void amplify() {
            Core.pow(image, 2, image);
        }

        /**
         * Wendet einen Boxfilter mit der angegegebenen Größe an
         * @param size Größe des Filterkerns
         */
        public void boxFilter(int size) {
            Imgproc.boxFilter(image, image, -1, new Size(size,size));
        }

        /**
         * Entfernt den Hintergrund aus dem Bild
         */
        public void removeBackground() {
            if(frameCount > 5) {
                if(bgImage.empty()) {
                    this.recaptureBackground();
                    System.out.println("recaptured");
                }
                Core.absdiff(image, bgImage, image);
            }
        }

        /**
         * Speichert das aktuelle Bild als Hintergrundbild
         */
        public void recaptureBackground() {
            image.copyTo(bgImage);
        }

        /**
         * Wendet einen binären Hochpassfilter an
         * @param minValue Minimaler Wert, ab dem die Grauwerte durch Weiß ersetzt
werden sollen
         */
        public void binaryThreshold(int minValue) {
            Imgproc.threshold(image, image,minValue, 255, Imgproc.THRESH_BINARY);
        }

        /**
```

```
	 * Wendet einen Hochpassfilter an
	 * @param minValue Minimaler Wert, ab dem die Grauwerte durch Weiß ersetzt
werden sollen
	 */
	public void threshold(int minValue) {
		Imgproc.threshold(image, image,minValue, 255, Imgproc.THRESH_TOZERO);
	}
}
```

Klasse – ShaderManager

```
package managers;

import com.jogamp.common.nio.Buffers;
import com.jogamp.opengl.GL4;
import com.jogamp.opengl.util.glsl.ShaderUtil;

public class ShaderManager {

	private static ShaderManager instance;
	public static ShaderManager getInstance() {
		if (instance == null) {
			instance = new ShaderManager();
		}
		return instance;
	}

	/**
	 * Erstellt einen Shader Manager
	 */
	private ShaderManager() {};

	private GL4 gl;

	/* Shader Programm 1 : Diffuser Farbshader */
	public int SHADER_DIFFUSE;

	String[] vertex_shader1 = {
			"#version 400\n"+
			"layout(location = 0) in vec3 vertex_position;"+
			"layout(location = 1) in vec3 vertex_colour;"+
			"layout(location = 2) in vec3 vertex_normal;"+
			"out vec3 frag_colour;"+
			"out vec3 frag_vertex;"+
			"out vec3 frag_normal;"+
			"uniform mat4 transf;"+
			"uniform mat4 projection;"+
			"uniform mat4 view;"+
			"void main () {"+
			"frag_colour = vertex_colour;"+
			"frag_normal = vertex_normal;"+
			"frag_vertex = vertex_position;"+
			"gl_Position = projection * view * transf *
vec4(vertex_position, 1.0);"+
```

```
                "}"
        };

        String[] fragment_shader1 = {
                "#version 400\n"+
                "in vec3 frag_colour;"+
                "in vec3 frag_vertex;"+
                "in vec3 frag_normal;"+
                "uniform vec3 lightPos;"+
                "uniform float lightInt;"+
                "uniform mat4 transf;"+
                "out vec4 final_colour;"+
                "void main () {"+
                "mat3 normalMatrix = transpose(inverse(mat3(transf)));"+
              "vec3 normal = normalize(normalMatrix * frag_normal);"+
              "vec3 frag_position = vec3(transf * vec4(frag_vertex, 1));"+
              "vec3 surfaceToLight = lightPos - frag_position;"+
              "float brightness = dot(normal, surfaceToLight) /
(length(surfaceToLight) * length(normal));"+
              "brightness = clamp(brightness, 0, 1);"+
              "final_colour = vec4(brightness * lightInt * frag_colour, 1.0);"+
                "}"
        };


        /**
         * Setzt die GL4 Referenz
         * @param gl Die zu setzende Referenz
         */
        public void setGL(GL4 gl) {
                this.gl = gl;
        }

        /**
         * Kompiliert die Shader und erstellt die benötigten Shader-Programme
         */
        public void buildShaders() {
                /* Shader Programm 1 : Grundlegende Farbshader */
                int vs = gl.glCreateShader(GL4.GL_VERTEX_SHADER);
                gl.glShaderSource(vs, 1, vertex_shader1,
Buffers.newDirectIntBuffer(new int[] {0}));
                gl.glCompileShader(vs);

                int fs = gl.glCreateShader (GL4.GL_FRAGMENT_SHADER);
                gl.glShaderSource(fs, 1, fragment_shader1,
Buffers.newDirectIntBuffer(new int[] {0}));
                gl.glCompileShader(fs);

                SHADER_DIFFUSE = gl.glCreateProgram();
                gl.glAttachShader(SHADER_DIFFUSE, fs);
                gl.glAttachShader(SHADER_DIFFUSE, vs);
                gl.glLinkProgram(SHADER_DIFFUSE);
        }
}
```

Klasse – TouchPoint

```java
package utils;

import org.opencv.core.Point;

public class TouchPoint extends Point {
	private int size;

	/**
	 * Erstellt einen Touchpunkt aus einem vorhandenen Punkt
	 * @param p Bereits vorhandener Punkt
	 * @param size Größe des Punktes
	 */
	public TouchPoint(Point p, int size) {
		super(p.x, p.y);
		this.size = size;
	}

	/**
	 * Erstellt einen Touchpunkt aus den Koordinaten
	 * @param x X-Koordinate des Punktes
	 * @param y Y-Koordinate des Punktes
	 * @param size Größe des Punktes
	 */
	public TouchPoint(int x, int y, int size) {
		super(x,y);
		this.size = size;
	}

	/**
	 * Gibt die Größe des Touchpunktes zurück
	 * @return Größe des Touchpunktes
	 */
	public int getSize() {
		return size;
	}
}
```

Klasse – Farbe

```java
package utils;

public class Farbe {
	public static final int WEISS = 0,
				SCHWARZ = 1,
				ROT = 2,
				BLAU = 3;
}
```

Klasse – Geometry

```
package geom;

import java.nio.FloatBuffer;
import javax.vecmath.Vector3f;

import com.jogamp.common.nio.Buffers;
import com.jogamp.opengl.GL;
import com.jogamp.opengl.GL4;
import com.jogamp.opengl.math.Matrix4;

public abstract class Geometry {
	protected GL4 gl;

	protected Matrix4 transform;
	protected Vector3f velocity, location, rotation, scale;
	protected float collisionRadius, mass, friction, colVelMulti;

	protected int shader_prog;

	/**
	 * Oberklassen Konstruktor um Geometrieobjekte zu beschreiben
	 * @param gl GL4 Objektreferenz
	 * @param shader_prog ID des zu benutzenden Shader Programms
	 * @param locX Position in X-Richtung
	 * @param locY Position in Y-Richtung
	 * @param locZ Position in Z-Richtung
	 * @param scaleX Skalierung in X-Richtung
	 * @param scaleY Skalierung in Y-Richtung
	 * @param scaleZ Skalierung in Z-Richtung
	 */
	protected Geometry(GL4 gl, int shader_prog, float locX, float locY, float
locZ, float scaleX, float scaleY, float scaleZ) {
		this(gl,shader_prog,locX, locY, locZ, 0, 0, 0, scaleX, scaleY,
scaleZ);
	}

	/**
	 * Oberklassen Konstruktor um Geometrieobjekte zu beschreiben
	 * @param gl GL4 Objektreferenz
	 * @param shader_prog ID des zu benutzenden Shader Programms
	 * @param locX Position in X-Richtung
	 * @param locY Position in Y-Richtung
	 * @param locZ Position in Z-Richtung
	 * @param rotX Rotation um X-Achse
	 * @param rotY Rotation um Y-Achse
	 * @param rotZ Rotation um Z-Achse
	 * @param scaleX Skalierung in X-Richtung
	 * @param scaleY Skalierung in Y-Richtung
	 * @param scaleZ Skalierung in Z-Richtung
	 */
	protected Geometry(GL4 gl, int shader_prog, float locX, float locY, float
locZ, float rotX, float rotY, float rotZ, float scaleX, float scaleY, float
scaleZ) {
		this.gl = gl;
		this.shader_prog = shader_prog;
		this.transform = new Matrix4();
```

```
		this.velocity = new Vector3f();
		this.location = new Vector3f();
		this.rotation = new Vector3f();
		this.scale = new Vector3f();

		this.location.x = locX;
		this.location.y = locY;
		this.location.z = locZ;
		this.rotation.x = rotX;
		this.rotation.y = rotY;
		this.rotation.z = rotZ;
		this.scale.x = scaleX;
		this.scale.y = scaleY;
		this.scale.z = scaleZ;

		this.rebuildMatrix();
	}

	/**
	 * Erstellt ein VAO und dazugehörige VBOs aus den gegebenen Arrays und gibt dessen ID zurück
	 * @param points Array von Vertex-Positionsdaten
	 * @param normals Array von Vertex-Normalendaten
	 * @param colors Array von Vertex-Farbdaten
	 * @return
	 */
	protected int[] buildVAO(float[] points, float[] normals, float[] colors) {
		int[] points_vbo = new int[1];
		gl.glGenBuffers(1, points_vbo, 0);
		gl.glBindBuffer(GL.GL_ARRAY_BUFFER, points_vbo[0]);
		gl.glBufferData(GL.GL_ARRAY_BUFFER, points.length * Buffers.SIZEOF_FLOAT, FloatBuffer.wrap(points), GL.GL_STATIC_DRAW);

		int[] colours_vbo = new int[1];
		gl.glGenBuffers(1, colours_vbo, 0);
		gl.glBindBuffer(GL.GL_ARRAY_BUFFER, colours_vbo[0]);
		gl.glBufferData(GL.GL_ARRAY_BUFFER, colors.length * Buffers.SIZEOF_FLOAT, FloatBuffer.wrap(colors), GL.GL_STATIC_DRAW);

		int[] normals_vbo = new int[1];
		gl.glGenBuffers(1, normals_vbo, 0);
		gl.glBindBuffer(GL.GL_ARRAY_BUFFER, normals_vbo[0]);
		gl.glBufferData(GL.GL_ARRAY_BUFFER, normals.length * Buffers.SIZEOF_FLOAT, FloatBuffer.wrap(normals), GL.GL_STATIC_DRAW);

		int[] vao = new int[1];
		gl.glGenVertexArrays(1, vao, 0);
		gl.glBindVertexArray(vao[0]);
		gl.glBindBuffer(GL.GL_ARRAY_BUFFER, points_vbo[0]);
		gl.glVertexAttribPointer (0, 3, GL.GL_FLOAT, false, 0, 0);
		gl.glBindBuffer (GL.GL_ARRAY_BUFFER, colours_vbo[0]);
		gl.glVertexAttribPointer (1, 3, GL.GL_FLOAT, false, 0, 0);
		gl.glBindBuffer (GL.GL_ARRAY_BUFFER, normals_vbo[0]);
		gl.glVertexAttribPointer (2, 3, GL.GL_FLOAT, true, 0, 0);

		gl.glEnableVertexAttribArray (0);
		gl.glEnableVertexAttribArray (1);
		gl.glEnableVertexAttribArray (2);

		return vao;
	}
```

```
	/**
	 * Hilfsmethode, um ein Vertex-Farbarray einheitlich mit einer Farbe zu
füllen
	 * @param colors Zu füllendes Array
	 * @param r Rotanteil (0-255)
	 * @param g Grünanteil (0-255)
	 * @param b Blauanteil (0-255)
	 */
	protected void fillColors(float[] colors, int r, int g, int b) {
		for(int i = 0; i < colors.length; i+=3) {
			colors[i]= (float)r/255f;
			colors[i+1]= (float)g/255f;
			colors[i+2]= (float)b/255f;
		}
	}

	/**
	 * Rendert das Objekt. Diese Methode muss in der Unterklasse realisiert
werden
	 */
	public abstract void render();

	/**
	 * Methode um die Transformationsmatrix nach einer Transformation neu
aufzubauen
	 */
	protected void rebuildMatrix() {
		transform.loadIdentity();
		transform.translate(location.x, location.y, location.z);
		transform.rotate(rotation.x, 1, 0, 0);
		transform.rotate(rotation.y, 0, 1, 0);
		transform.rotate(rotation.z, 0, 0, 1);
		transform.scale(scale.x, scale.y, scale.z);
	}

	/**
	 * Überprüft ob dieses Objekt mit dem gegebenen anderen Geometrieobjekt g
kollidiert,
	 * dies wird anhand einer Kreiskollision berechnet
	 * @param g Das andere zu kollidierende Geometrieobjekt
	 */
	public void circleCollision(Geometry g) {
		Vector3f thisLoc = new Vector3f(location);
		Vector3f otherLoc = new Vector3f(g.getLocation());

		Vector3f thisVel = new Vector3f(velocity);
		Vector3f otherVel = new Vector3f(g.getVelocity());

		//thisLoc is now a vector from this ball to the other ball
		thisLoc.sub(otherLoc);

		//They Collide
		if(thisLoc.length() <= collisionRadius + g.getCollisionRadius()) {

			Vector3f normal = new Vector3f(g.location);
		normal.sub(location);
		normal.normalize();

		Vector3f rVel = new Vector3f(otherVel);
		rVel.sub(thisVel);
```

```
			float nVel = rVel.dot(normal);
				float a1 = thisVel.dot(normal);
				float a2 = otherVel.dot(normal);

				//Prevent "sticking" (constant recollision every frame) of
objects
				if(nVel <= 0) {
					//Calculate new velocity vectors
					float p = (float)(2.0 * (a1 - a2)) / (this.mass +
g.mass);

					velocity.x -= p * g.mass * normal.x * colVelMulti;
					velocity.z -= p * g.mass * normal.z * colVelMulti;

					g.velocity.x += p * this.mass * normal.x * colVelMulti;
					g.velocity.z += p * this.mass * normal.z * colVelMulti;
				}
			}
		}


		/**
		 * Überprüft, ob dieses Objekt mit einer der Wände kollidiert
		 * und kehrt den Bewegungsvektor um, falls dies der Fall ist
		 */
		public void wallCollision() {
			// Kugel befindet sich am Rand des Spielfelds? (links, rechts)
			if(Math.abs(location.x) + scale.x >= 0.635f) {
				//Kugel prallt vor Wand (links, rechts)
				if(Math.abs(location.z) < scale.z/2 + 1.155f) {
					//Kugel rollt nicht ins mittlere Loch
					if(Math.abs(location.z) > 0.065f - scale.z/2)  {
						//Wenn Kugel am rechten Rand
						if(location.x > 0) {
							velocity.x = -Math.abs(velocity.x);
						}
						//Sonst
						else {
							velocity.x = Math.abs(velocity.x);
						}
					}
				}
			// Kugel befindet sich am Rand des Spielfelds? (oben, unten)
			} else if(Math.abs(location.z) + scale.z >= 1.27f) {
				//Kugel prallt vor Wand? (oben, unten)
				if (Math.abs(location.x) < scale.x/2 + 0.52f) {
					//Wenn Kugel am unteren Rand
					if(location.z > 0) {
						velocity.z = -Math.abs(velocity.z);
					}
					//Sonst
					else {
						velocity.z = Math.abs(velocity.z);
					}
				}
			}
		}


		/**
```

```java
	 * Gibt die Position des Objekts zurück
	 * @return Position des Objekts
	 */
	public Vector3f getLocation() {
		return location;
	}

	/**
	 * Gibt die Rotation des Objekts zurück
	 * @return Rotation des Objekts
	 */
	public Vector3f getRotation() {
		return rotation;
	}

	/**
	 * Gibt die Skalierung des Objekts zurück
	 * @return Skalierung des Objekts
	 */
	public Vector3f getScale() {
		return scale;
	}

	/**
	 * Gibt den Bewegungsvektor des Objekts zurück
	 * @return Bewegungsvektor des Objekts
	 */
	public Vector3f getVelocity() {
		return velocity;
	}

	/**
	 * Gibt den Kollisionsradius zurück
	 * @return Kollisionsradius
	 */
	public float getCollisionRadius() {
		return collisionRadius;
	}

	/**
	 * Setzt den Kollisionsradius auf einen gegebenen Wert
	 * @param collisionRadius Neuer Kollisionsradius
	 */
	public void setCollisionRadius(float collisionRadius) {
		this.collisionRadius = collisionRadius;
	}

	/**
	 * Gibt die Masse des Objekts zurück
	 * @return Masse des Objekts
	 */
	public float getMass() {
		return mass;
	}

	/**
	 * Setzt die Masse auf einen neuen Wert
	 * @param mass Neue Masse des Objekts
	 */
	public void setMass(float mass) {
		this.mass = mass;
```

```
	}

	/**
	 * Bewegt das Objekt in einer flüssigen Bewegung um die angegebenen Werte
(pro Sekunde)
	 * @param locX Bewegung in X-Richtung pro Sekunde
	 * @param locY Bewegung in Y-Richtung pro Sekunde
	 * @param locZ Bewegung in Z-Richtung pro Sekunde
	 * @param delta Framezeit für die flüssige Animation
	 */
	public void move(float locX, float locY, float locZ, float delta) {
		this.location.x += locX * delta;
		this.location.y += locY * delta;
		this.location.z += locZ * delta;
		this.rebuildMatrix();
	}

	/**
	 * Bewegt das Objekt direkt in einem Sprung zum angegebenen Ort
	 * @param locX Neue X-Koordinate
	 * @param locY Neue Y-Koordinate
	 * @param locZ Neue Z-Koordinate
	 */
	public void moveTo(float locX, float locY, float locZ) {
		this.location.x = locX;
		this.location.y = locY;
		this.location.z = locZ;
		this.rebuildMatrix();
	}

	/**
	 * Rotiert das Objekt in einer flüssigen Bewegung um die angegebenen Werte
(pro Sekunde)
	 * @param rotX Rotation um X-Achse pro Sekunde
	 * @param rotY Rotation um Y-Achse pro Sekunde
	 * @param rotZ Rotation um Z-Achse pro Sekunde
	 * @param delta Framezeit für die flüssige Animation
	 */
	public void rotate(float rotX, float rotY, float rotZ, float delta) {
		this.rotation.x += (rotX * delta);
		this.rotation.y += (rotY * delta);
		this.rotation.z += (rotZ * delta);
		this.rebuildMatrix();
	}

	/**
	 * Rotiert das Objekt direkt zu den angegebenen absoluten Werten
	 * @param rotX Neue Rotation um X-Achse
	 * @param rotY Neue Rotation um Y-Achse
	 * @param rotZ Neue Rotation um Z-Achse
	 */
	public void rotateTo(float rotX, float rotY, float rotZ) {
		this.rotation.x = rotX;
		this.rotation.y = rotY;
		this.rotation.z = rotZ;
		this.rebuildMatrix();
	}

	/**
	 * Skaliert das Objekt in einer flüssigen Animation um die angegebenen Werte
	 * @param scaleX Skalierung in X-Richtung pro Sekunde
```

```
 * @param scaleY Skalierung in Y-Richtung pro Sekunde
 * @param scaleZ Skalierung in Z-Richtung pro Sekunde
 * @param delta Framezeit für die flüssige Animation
 */
public void scale(float scaleX, float scaleY, float scaleZ, float delta) {
	this.scale.x += scaleX * delta;
	this.scale.y += scaleY * delta;
	this.scale.z += scaleZ * delta;
	this.rebuildMatrix();
}

/**
 * Skaliert das Objekt direkt zu der angegebenen absoluten Skalierung
 * @param scaleX Neue Skalierung in X-Richtung
 * @param scaleY Neue Skalierung in Y-Richtung
 * @param scaleZ Neue Skalierung in Z-Richtung
 */
public void scaleTo(float scaleX, float scaleY, float scaleZ) {
	this.scale.x = scaleX;
	this.scale.y = scaleY;
	this.scale.z = scaleZ;
	this.rebuildMatrix();
}

/**
 * Setzt den Bewegungsvektor neu
 * @param velocity Neuer Bewegungsvektor
 */
public void setVelocity(Vector3f velocity) {
	this.velocity = velocity;
}

/**
 * Setzt den Bewegungsvektor neu
 * @param velX Neue Bewegungsgeschwindigkeit in X-Richtung
 * @param velY Neue Bewegungsgeschwindigkeit in Y-Richtung
 * @param velZ Neue Bewegungsgeschwindigkeit in Z-Richtung
 */
public void setVelocity(float velX, float velY, float velZ) {
	this.velocity.x = velX;
	this.velocity.y = velY;
	this.velocity.z = velZ;
}

/**
 * Addiert den angegebenen Vektor mit dem aktuellen Geschwindigkeitsvektor
 * @param velocity Der zu Addierende Geschwindigkeitsvektor
 */
public void addVelocity(Vector3f velocity) {
	this.velocity.add(velocity);
}

/**
 * Addiert den angegebenen Vektor mit dem aktuellen Geschwindigkeitsvektor
 * @param velX Die zu addierende Bewegungsgeschwindigkeit in X-Richtung
 * @param velY Die zu addierende Bewegungsgeschwindigkeit in Y-Richtung
 * @param velZ Die zu addierende Bewegungsgeschwindigkeit in Z-Richtung
 */
public void addVelocity(float velX, float velY, float velZ) {
	this.velocity.x += velX;
	this.velocity.y += velY;
```

```
			this.velocity.z += velZ;
		}

		/**
		 * Wendet den Bewegungsvektor des Objekts auf seine Position an,
		 * womit sich das Objekt pro Sekunde um die im Vektor gegebenen Werte bewegt
		 * @param delta Framezeit für die flüssige Animation
		 */
		public void applyVelocity(float delta) {
			//Calculate per-frame friction and per-frame velocity
			//Friction is a percent value applied over a time of one second
			Vector3f frictionVector = new Vector3f(velocity);
			frictionVector.scale(delta);
			Vector3f scaledVel = new Vector3f(frictionVector);
			frictionVector.scale(friction);

			//Subtract friction from object velocity
			velocity.sub(frictionVector);

			//Apply per-frame velocity to current location
			this.location.add(scaledVel);
			this.rebuildMatrix();
		}
}
```

Klasse – Kugel

```
package geom;

import java.util.ArrayList;
import com.jogamp.opengl.GL4;

import utils.Farbe;

public class Kugel extends Geometry {

		private int colour;
		private boolean deletionMarked;
		private int decay = 1000, curDecay = 0;

		private int[] vao;

		private static float X = 0.525731112119133606f;
		private static float Z = 0.850650808352039932f;

		private float[] vertices;
		private ArrayList<Float> tempVertices;

		private float[][] icovertices = {
			{-X,0.0f,Z},
			{X,0.0f,Z},
			{-X,0.0f,-Z},
			{X,0.0f,-Z},
			{0.0f,Z,X},
```

```
            {0.0f,Z,-X},
            {0.0f,-Z,X},
            {0.0f,-Z,-X},
            {Z,X,0.0f},
            {-Z,X,0.0f},
            {Z,-X,0.0f},
            {-Z,-X,0.0f}
      };

      private int[][] icoindices = {
            { 0, 4, 1 },
            { 0, 9, 4 },
            { 9, 5, 4 },
        { 4, 5, 8 },
        { 4, 8, 1 },
        { 8, 10, 1 },
        { 8, 3, 10 },
        { 5, 3, 8 },
        { 5, 2, 3 },
        { 2, 7, 3 },
        { 7, 10, 3 },
        { 7, 6, 10 },
        { 7, 11, 6 },
        { 11, 0, 6 },
        { 0, 1, 6 },
        { 6, 1, 10 },
        { 9, 0, 11 },
        { 9, 11, 2 },
        { 9, 2, 5 },
        { 7, 2, 11 }
    };

      private float[] colors = {};

      public Kugel(GL4 gl, int shader_prog, float locX, float locY, float locZ,
float scale, int color) {
            super(gl, shader_prog, locX, locY, locZ, scale, scale, scale);
            buildSphere(2);


            this.colour = color;
            colors = new float[vertices.length];
            switch(color) {
                  case Farbe.WEISS:
                        fillColors(colors, 255, 255, 255); break;
                  case Farbe.SCHWARZ:
                        fillColors(colors, 20, 20, 20); break;
                  case Farbe.ROT:
                        fillColors(colors, 248, 45, 39); break;
                  case Farbe.BLAU:
                        fillColors(colors, 53, 64, 231); break;
            }
            vao = buildVAO(vertices, vertices, colors);

            this.collisionRadius = scale;
            this.mass = 4;
            this.friction = 0.35f;
            this.colVelMulti = 0.8f;
      }

      /**
```

```java
 * Trianguliert die ursprünglichen 12 Flächen des Ikosaeder,
 * wodurch eine ideale Kugelform weiter angenähert wird
 * @param depth Rekursionstiefe für die Triangulierung
 */
private void buildSphere(int depth) {
      tempVertices = new ArrayList<Float>();

  for (int i = 0; i < 20; ++i) {
      subdivide(
        icovertices[icoindices[i][0]],
        icovertices[icoindices[i][1]],
        icovertices[icoindices[i][2]],
        depth);
  }

  vertices = new float[tempVertices.size()];
  for(int i = 0; i < tempVertices.size(); i++) {
      vertices[i] = tempVertices.get(i);
  }
}

/**
 * Unterteilt ein gegebenes Dreieck
 * @param v1 Vertex 1 der Fläche
 * @param v2 Vertex 2 der Fläche
 * @param v3 Vertex 3 der Fläche
 * @param depth Rekursionstiefe
 */
private void subdivide(float v1[], float v2[], float v3[], int depth) {
      float[] v12 = new float[3];
      float[] v23 = new float[3];
      float[] v31 = new float[3];

      if (depth==0) {
              tempVertices.add(v1[0]);
              tempVertices.add(v1[1]);
              tempVertices.add(v1[2]);

              tempVertices.add(v2[0]);
              tempVertices.add(v2[1]);
              tempVertices.add(v2[2]);

              tempVertices.add(v3[0]);
              tempVertices.add(v3[1]);
              tempVertices.add(v3[2]);
            return;
      }
      for (int i=0; i<3; i++) {
            v12[i] = (v1[i]+v2[i])/2.0f;
            v23[i] = (v2[i]+v3[i])/2.0f;
            v31[i] = (v3[i]+v1[i])/2.0f;
      }

      v12 = normalize(v12);
      v23 = normalize(v23);
      v31 = normalize(v31);

      subdivide(v2,v23,v12,depth-1);
      subdivide(v1,v12,v31,depth-1);
      subdivide(v3,v31,v23,depth-1);
      subdivide(v12,v23,v31,depth-1);
```

```
}

/**
 * Normiert den Vektor
 * @param v Der zu normierende Vektor
 * @return Der neue normierte Vektor
 */
private float[] normalize(float v[]){
   float len = 0;
   for(int i = 0; i < 3; ++i){
           len += v[i] *  v[i];
   }
   len = (float) Math.sqrt(len);
   for(int i = 0; i < 3; ++i){
           v[i] /= len;
   }
   return v;
 }

/**
 * Gibt die Farbe der Kugel zurück (Die Farbe im Spiel,
 * entspricht in diesem Fall den Vertexfarben)
 * @return Die Konstante, welche die Farbe der Kugel angibt
 */
public int getColour() {
      return colour;
}

/**
 * Merke eine Kugel zur Löschung vor
 */
public void markForDeletion() {
      deletionMarked = true;
}

/**
 * Überprüft, ob die aktuelle Lebenszeit nach der Vermerkung
 * die erlaubte Lebenszeit überschritten hat
 * @param lastMillis Die aktuelle Framezeit
 * @return Boolean, der angibt, ob das Objekt gelöscht werden soll
 */
public boolean checkForDeletion(float lastMillis) {
      if(deletionMarked) {
            curDecay += lastMillis;
            if(curDecay >= decay) {
                  return true;
            }
      }
      return false;
}

/**
 * Rendert die Kugel
 */
@Override
public void render() {
      gl.glUseProgram(shader_prog);
            int transLoc = gl.glGetUniformLocation(shader_prog, "transf");
      gl.glUniformMatrix4fv(transLoc, 1, false, transform.getMatrix(),0);
      gl.glBindVertexArray(vao[0]);
      gl.glDrawArrays(GL4.GL_TRIANGLES, 0, vertices.length/3);
```

```
	}

	/**
	 * Gibt an, ob die Kugel zum Löschen vorgemerkt wurde
	 * @return Boolean, ob dies zutrifft
	 */
	public boolean isMarkedForDeletion() {
		return deletionMarked;
	}

	/**
	 * Setzt die gezählte Lebenszeit zurück
	 */
	public void resetDecay() {
		curDecay = 0;
	}
}
```

Klasse – Tisch

```
package geom;

import com.jogamp.opengl.GL4;

public class Tisch extends Geometry {

	private int[] vao;

	private float[] vertices = {
			//Grundfläche
			-0.635f, 0.0f, 1.27f,
			0.635f, 0.0f, 1.27f,
			0.635f, 0.0f, -1.27f,
			-0.635f, 0.0f, -1.27f,
			//Vertikale Seitenkanten
				//Kante unten links
				-0.635f, 0.0f, 1.155f,
				-0.635f, 0.0f, 0.065f,
				-0.635f, 0.038f, 0.065f,
				-0.635f, 0.038f, 1.155f,
				//Kante oben links
				-0.635f, 0.0f, -0.065f,
				-0.635f, 0.0f, -1.155f,
				-0.635f, 0.038f, -1.155f,
				-0.635f, 0.038f, -0.065f,
				//Kante oben mitte
				-0.52f, 0.0f, -1.27f,
				0.52f, 0.0f, -1.27f,
```

```
                0.52f, 0.038f, -1.27f,
                -0.52f, 0.038f, -1.27f,
                //Kante oben rechts
                0.635f, 0.0f, -1.155f,
                0.635f, 0.0f, -0.065f,
                0.635f, 0.038f, -0.065f,
                0.635f, 0.038f, -1.155f,
                //Kante unten rechts
                0.635f, 0.0f, 0.065f,
                0.635f, 0.0f, 1.155f,
                0.635f, 0.038f, 1.155f,
                0.635f, 0.038f, 0.065f,
                //Kante oben mitte
                0.52f, 0.0f, 1.27f,
                -0.52f, 0.0f, 1.27f,
                -0.52f, 0.038f, 1.27f,
                0.52f, 0.038f, 1.27f,

            //Horizontale Seitenkanten
                //Kante unten links
                -0.635f, 0.038f, 1.155f,
                -0.635f, 0.038f, 0.065f,
                -0.735f, 0.038f, 0.065f,
                -0.735f, 0.038f, 1.155f,
                //Kante oben links
                -0.635f, 0.038f, -0.065f,
                -0.635f, 0.038f, -1.155f,
                -0.735f, 0.038f, -1.155f,
                -0.735f, 0.038f, -0.065f,
                //Kante oben mitte
                -0.52f, 0.038f, -1.27f,
                0.52f, 0.038f, -1.27f,
                0.52f, 0.038f, -1.37f,
                -0.52f, 0.038f, -1.37f,
                //Kante oben rechts
                0.635f, 0.038f, -1.155f,
                0.635f, 0.038f, -0.065f,
                0.735f, 0.038f, -0.065f,
                0.735f, 0.038f, -1.155f,
                //Kante unten rechts
                0.635f, 0.038f, 0.065f,
                0.635f, 0.038f, 1.155f,
                0.735f, 0.038f, 1.155f,
                0.735f, 0.038f, 0.065f,
                //Kante unten mitte
                0.52f, 0.038f, 1.27f,
                -0.52f, 0.038f, 1.27f,
                -0.52f, 0.038f, 1.37f,
                0.52f, 0.038f, 1.37f
};

private float[] normals = {
            //Grundfläche
            0.0f, 1.0f, 0.0f,
            0.0f, 1.0f, 0.0f,
            0.0f, 1.0f, 0.0f,
            0.0f, 1.0f, 0.0f,
            //Vertikale Seitenkanten
                //Kante unten links
                1.0f, 0.0f, 0.0f,
                1.0f, 0.0f, 0.0f,
```

```
                    1.0f, 0.0f, 0.0f,
                    1.0f, 0.0f, 0.0f,
                    //Kante oben links
                    1.0f, 0.0f, 0.0f,
                    1.0f, 0.0f, 0.0f,
                    1.0f, 0.0f, 0.0f,
                    1.0f, 0.0f, 0.0f,
                    //Kante oben mitte
                    0.0f, 0.0f, 1.0f,
                    0.0f, 0.0f, 1.0f,
                    0.0f, 0.0f, 1.0f,
                    0.0f, 0.0f, 1.0f,
                    //Kante oben rechts
                    -1.0f, 0.0f, 0.0f,
                    -1.0f, 0.0f, 0.0f,
                    -1.0f, 0.0f, 0.0f,
                    -1.0f, 0.0f, 0.0f,
                    //Kante unten rechts
                    -1.0f, 0.0f, 0.0f,
                    -1.0f, 0.0f, 0.0f,
                    -1.0f, 0.0f, 0.0f,
                    -1.0f, 0.0f, 0.0f,
                    //Kante oben mitte
                    0.0f, 0.0f, -1.0f,
                    0.0f, 0.0f, -1.0f,
                    0.0f, 0.0f, -1.0f,
                    0.0f, 0.0f, -1.0f,

                //Horizontale Seitenkanten
                    //Kante unten links
                    0.0f, 1.0f, 0.0f,
                    0.0f, 1.0f, 0.0f,
                    0.0f, 1.0f, 0.0f,
                    0.0f, 1.0f, 0.0f,
                    //Kante oben links
                    0.0f, 1.0f, 0.0f,
                    0.0f, 1.0f, 0.0f,
                    0.0f, 1.0f, 0.0f,
                    0.0f, 1.0f, 0.0f,
                    //Kante oben mitte
                    0.0f, 1.0f, 0.0f,
                    0.0f, 1.0f, 0.0f,
                    0.0f, 1.0f, 0.0f,
                    0.0f, 1.0f, 0.0f,
                    //Kante oben rechts
                    0.0f, 1.0f, 0.0f,
                    0.0f, 1.0f, 0.0f,
                    0.0f, 1.0f, 0.0f,
                    0.0f, 1.0f, 0.0f,
                    //Kante unten rechts
                    0.0f, 1.0f, 0.0f,
                    0.0f, 1.0f, 0.0f,
                    0.0f, 1.0f, 0.0f,
                    0.0f, 1.0f, 0.0f,
                    //Kante unten mitte
                    0.0f, 1.0f, 0.0f,
                    0.0f, 1.0f, 0.0f,
                    0.0f, 1.0f, 0.0f,
                    0.0f, 1.0f, 0.0f
};
```

```
private float[] colors = {
            //Grundfläche
            0.024f, 0.643f, 0.302f,
            0.024f, 0.643f, 0.302f,
            0.024f, 0.643f, 0.302f,
            0.024f, 0.643f, 0.302f,
            //Vertikale Seitenkanten
                  //Kante unten links
                  0.024f, 0.643f, 0.302f,
                  0.024f, 0.643f, 0.302f,
                  0.024f, 0.643f, 0.302f,
                  0.024f, 0.643f, 0.302f,
                  //Kante oben links
                  0.024f, 0.643f, 0.302f,
                  0.024f, 0.643f, 0.302f,
                  0.024f, 0.643f, 0.302f,
                  0.024f, 0.643f, 0.302f,
                  //Kante oben mitte
                  0.024f, 0.643f, 0.302f,
                  0.024f, 0.643f, 0.302f,
                  0.024f, 0.643f, 0.302f,
                  0.024f, 0.643f, 0.302f,
                  //Kante oben rechts
                  0.024f, 0.643f, 0.302f,
                  0.024f, 0.643f, 0.302f,
                  0.024f, 0.643f, 0.302f,
                  0.024f, 0.643f, 0.302f,
                  //Kante unten rechts
                  0.024f, 0.643f, 0.302f,
                  0.024f, 0.643f, 0.302f,
                  0.024f, 0.643f, 0.302f,
                  0.024f, 0.643f, 0.302f,
                  //Kante oben mitte
                  0.024f, 0.643f, 0.302f,
                  0.024f, 0.643f, 0.302f,
                  0.024f, 0.643f, 0.302f,
                  0.024f, 0.643f, 0.302f,
            //Horizontale Seitenkanten
                  //Kante unten links
                  0.216f, 0.066f, 0.066f,
                  0.216f, 0.066f, 0.066f,
                  0.216f, 0.066f, 0.066f,
                  0.216f, 0.066f, 0.066f,
                  //Kante oben links
                  0.216f, 0.066f, 0.066f,
                  0.216f, 0.066f, 0.066f,
                  0.216f, 0.066f, 0.066f,
                  0.216f, 0.066f, 0.066f,
                  //Kante oben mitte
                  0.216f, 0.066f, 0.066f,
                  0.216f, 0.066f, 0.066f,
                  0.216f, 0.066f, 0.066f,
                  0.216f, 0.066f, 0.066f,
                  //Kante oben rechts
                  0.216f, 0.066f, 0.066f,
                  0.216f, 0.066f, 0.066f,
                  0.216f, 0.066f, 0.066f,
                  0.216f, 0.066f, 0.066f,
                  //Kante unten rechts
                  0.216f, 0.066f, 0.066f,
                  0.216f, 0.066f, 0.066f,
```

```
                        0.216f, 0.066f, 0.066f,
                        0.216f, 0.066f, 0.066f,
                        //Kante unten mitte
                        0.216f, 0.066f, 0.066f,
                        0.216f, 0.066f, 0.066f,
                        0.216f, 0.066f, 0.066f,
                        0.216f, 0.066f, 0.066f
        };

        public Tisch(GL4 gl, int shader_prog, float locX, float locY, float locZ,
float scaleX, float scaleY, float scaleZ) {
                super(gl, shader_prog, locX, locY, locZ, scaleX, scaleY, scaleZ);
                vao = buildVAO(vertices, normals, colors);
        }

        public Tisch(GL4 gl, int shader_prog, float locX, float locY, float locZ,
float rotX, float rotY, float rotZ, float scaleX, float scaleY, float scaleZ) {
                super(gl, shader_prog, locX, locY, locZ, rotX, rotY, rotZ, scaleX,
scaleY, scaleZ);
                vao = buildVAO(vertices, normals, colors);
        }

        /**
         * Rendert den Tisch
         */
        @Override
        public void render() {
                gl.glUseProgram(shader_prog);
                        int transLoc = gl.glGetUniformLocation(shader_prog, "transf");
                gl.glUniformMatrix4fv(transLoc, 1, false, transform.getMatrix(),0);
                gl.glBindVertexArray(vao[0]);
                gl.glDrawArrays(GL4.GL_QUADS, 0, 84);
        }

}
```