BunnySuite: Ein Framework für 2D-Grafik-Performance-Tests

Ausarbeitung zum Serious Games Praktikum

Maximilian Li, Victor Ferdinand Schümmer, Patrick Pauli

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Elektrotechnik
und Informationstechnik
Fachbereich Informatik (Zweitmitglied)

Fachgebiet Multimedia Kommunikation
Prof. Dr.-Ing. Ralf Steinmetz

Ehrenwörtliche Erklärung

Hiermit versicheren wir, die vorliegende Ausarbeitung zum Serious Games Praktikum ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in dieser oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Die schriftliche Fassung stimmt mit der elektronischen Fassung überein.

Darmstadt, den 13. September 2016 Maximilian Li, Victor Ferdinand Schümmer, Patrick Pauli

Inhaltsverzeichnis

1 Einleitung **2**

2 Zeitplan und Aufgabenverteilung **3**

3 Grundlagen **4**
3.1 2D-Frameworks 4
3.2 Benchmarking 5

4 Konzept und Design **7**
4.1 Allgemeines 7
4.2 Identifizierte Bottlenecks 7

5 Umsetzung und Tests **8**
5.1 Entwicklungsablauf 8
5.2 Ablauf der Tests 8
5.3 Beschreibung der Tests 9
5.4 Implementierungsdetails bei den einzelnen Frameworks 13
5.4.1 Kha 13
5.4.2 SDL 13
5.4.3 XNA/Monogame 14
5.4.4 LibGDX 14
5.5 User Interface 14
5.5.1 GUI 14
5.5.2 Tests direkt starten 14

6 Ergebnisse der Messungen **16**
6.1 XNA Forced VSync? 16
6.2 SDL Primitive 18
6.3 Die Parameter Repetitions und Schrittweite 19

7 Zusammenfassung und Ausblick **20**

Literaturverzeichnis **20**

Zusammenfassung

Im Rahmen des Serious Games Praktikums wurde *BunnySuite* entwickelt, ein Benchmark-Framework für 2D-Grafik-Bibliotheken. Es besteht aus einer grafischen Benutzeroberfläche sowie den Implementierungen für die vier Bibliotheken Kha, LibGDX, SDL und XNA/Monogame. Die Tests werden in der GUI definiert und anschließend vollautomatisch durchgeführt. Die Ergebnisse werden grafisch in Diagrammen aufbereitet. Die Testergebnisse zeigen, dass die Leistungsfähigkeit der getesteten Bibliotheken in verschiedenen Bereichen sehr unterschiedlich sind. Dadurch, dass *BunnySuite* diese Unterschiede transparent macht, können Entwickler leichter die für ihren Zweck beste Bibliothek finden. Außerdem helfen die Ergebnisse den Entwicklern der Bibliotheken bei der Optimierung der Performance.

1 Einleitung

Moderne Computerspiele setzen meistens auf eine graphische Darstellung ihrer Spielwelt. Dabei geht, vor allem in den letzten Jahren, der Trend verstärkt in die Richtung fotorealistischer 3D-Renderings und weg von der simpleren 2D-Darstellung. Dabei ist schnelles 2D-Rendering immer noch eine nichttriviale Aufgabe für GPUs. Für viele Anwendungen sind die verfügbaren Bibliotheken im Allgemeinen immer noch viel zu langsam. Um das zu verbessern, haben wir mit *BunnySuite* ein Framework entwickelt, mit dem gängige 2D-Grafik-Bibliotheken einem automatisierten Stresstest ausgesetzt werden können.
Mit dem *BunnyMark*, entwickelt von Iain Lobb[1], gab es bereits eine Metrik zum Vergleich von Bibliotheken. Jedoch wurde hier nur die Anzahl der gerenderten Objekte (in diesem Fall Häschen, deshalb der Name) gemessen, bei denen noch 60 fps erreicht werden. Diese Zahl war nur schwer zu interpretieren und musste für ein ausführlicheres Ranking durch die aussagekräftigere Metrik *Renderzeit pro Frame bei X Objekten* ersetzt werden. Auch findet diese Messung bei BunnyMark zu Demozwecken nur interaktiv statt: Häschen werden per Mausklick zur Szene hinzugefügt. Im von uns entwickelten BunnySuite-Framework können solche Messungen automatisiert erfolgen. Die Tests für mehrere Frameworks werden automatisch nacheinander gestartet und das Ergebnis wird in zwei Diagrammen zusammengefasst. Das Framework ist leicht erweiterbar, so dass man es mit wenig Aufwand an neue Bibliotheken anpassen kann.
Das vollautomatisierte Test-Framework soll Entscheidern bei der Spieleentwicklung die Möglichkeit eines Rankings bieten, um die richtige Bibliothek für ihre Anforderungen zu finden. Desweiteren sollen Entwickler von Bibliotheken das Framework nutzen können, um ihre eigene Bibliothek zu testen und zu optimieren. Die Leistungen der Bibliotheken können so transparent verglichen werden, was den Wettbewerb zwischen ihnen stimulieren und Anreize setzen soll, stärker an der Performanz zu arbeiten.
Im folgenden Kapitel wird zunächst kurz der Projektplan aufgeführt (Kapitel 2) und anschließend werden die Grundlagen von 2D-Grafik-Libraries und Benchmarking beschrieben (Kapitel 3). Im Hauptteil der Ausarbeitung werden das grundlegende Konzept (Kapitel 4) und einige Aspekte der praktischen Umsetzung des entwickelten Frameworks (Kapitel 5) vorgestellt. Danach gehen wir in Kapitel 6 auf ein paar interessante Testergebnisse ein. Im letzten Kapitel wird die Arbeit zusammengefasst und es werden einige Erweiterungsmöglichkeiten genannt.

[1] `http://blog.iainlobb.com/2010/11/display-list-vs-blitting-results.html`

2 Zeitplan und Aufgabenverteilung

Das Team hat sich darauf geeinigt, dass jeder Entwickler sich auf eine Bibliothek konzentriert. In regelmäßigen Meetings werden Konzepte diskutiert und Designentscheidungen getroffen. Auf diese Weise soll in allem Frameworks eine vergleichbare Implementierung erreicht werden.

	1 PT = 8h		Wer?	
Arbeitspakete	Summe (Personentage)	Victor	Max	Patrick
1 Management und Koordination	6	2	2	2
1.1 Controlling	3	1	1	1
1.2 Koordination	3	1	1	1
2 Recherche	22	7	7	8
2.1 Grafik-Libraries einarbeiten	4	1	1	2
2.2 Einarbeitung in Bibliotheken	3	1	1	1
2.3 Recherche nach Bottlenecks	15	5	5	5
3 Konzept	18	6	6	6
3.1 Design des Frameworks	4	1	1	2
3.2 Design der Tests	14	5	5	4
4 Implementierung	20	7	7	6
4.1 Implementierung des Frameworks	4	1	1	2
4.2 Implementierung der Tests	16	6	6	4
4.2.1 Tests für LibGDX	4	4	0	0
4.2.2 Tests für SDL	4	0	0	4
4.2.3 Tests für kha	4	2	2	0
4.2.4 Tests für XNA Monogame	4	0	4	0
5 UI	2	1	1	0
6 Ausarbeitung	8	2	2	3
Gesamtaufwand	75	25	25	25

3 Grundlagen

3.1 2D-Frameworks

Game-Frameworks sollen die Entwickler beim Implementieren der Game-Loop unterstützen. Diese Game-Loop besteht hauptsächlich aus 2 Phasen:

Update() Der interne Zustand der Spielwelt und ihrer Objekte wird aktualisiert, beispielsweise die Position der Spielfigur gemäß der Benutzereingaben angepasst oder das nächste Frame in einer Animation wird ausgewählt oder die Punktzahl erhöht. Zeitgleich finden hier auch komplexere Berechnungen statt, wie die Kollisionsbestimmung, die wieder zu neuen Berechnungen führen kann (Positionen werden angepasst, damit Objekte sich nicht überlagern; "Gesundheitswerte" werden bei Treffern verringert).

Draw() Die im vorherigen Schritt berechnete Spielwelt wird auf dem Bildschirm dargestellt. Dazu gehören die korrekte Position aller Objekte, die Texturierung und Beleuchtung dieser Objekte, und gegebenenfalls Transformationen (Skalierung, Rotation). Auch muss bei Überlagerung von Objekten auf die korrekte Darstellung geachtet werden. Dies gilt auch für verdeckte Rückseiten von Objekten. Anschließend wird noch die Benutzeroberfläche über den Bildschirminhalt gelegt.

Die Game-Frameworks unterstützen die Entwickler, indem sie verschiedene Funktionalitäten bereits für die jeweilige Zielplatform optimiert zur Verfügung stellen. Dazu gehören:

- Einfacher Zugriff auf Eingabegeräte. Das können einfach Maus und Tastatur sein, aber auch plattformspezifische Gamepads oder auch Touch- und Gestensteuerung, vor allem für VR.

- optimierte Klassen für Assets. Dazu gehören Audiodateien, die gestreamt werden, oder auch Texturen, die möglichst effizient für den Grafikspeicher komprimiert werden.

- optimierte Renderfunktionen, die direkt Grafikschnittstellen wie OpenGL, Direct3D oder Vulkan ansprechen. Diese können auch bereits Transformationen wie Drehungen oder Einfärbungen anbieten. Diese können direkt 3D-Meshes, Bilder/Texturen, Text oder auch Primitive wie Dreiecke oder Rechtecke darstellen.

Diese Bibliotheken helfen den Entwicklern, eine effiziente Implementierung zu erreichen, ohne sehr hardwarenahe Optimierungen selber programmieren zu müssen. Somit soll sehr einfach der "Goldstandard" von 60 Frames per Second in der Darstellungen erreicht werden, bei dem für jede Iteration der Gameloop eine Rechenzeit von knapp 16,67 ms zur Verfügung steht. Dies beinhaltet das Einlesen von Assets wie Texturen vom Speichersystem der jeweiligen Plattform, was bereits für Verzögerungen sorgen kann.
Der wahrscheinlich größte Unterschied zu 3D-Frameworks besteht in der Komplexität der Grafikberechnung. So muss das 2D-Framework keine Informationen über Oberflächen der Objekte, z.B. Oberflächenorientierung zu Lichtquellen über NormalMaps oder Materialeigenschaften für das *Physically Based Rendering (PBR)*, außer der konkreten Farbe, kennen.
Auch können diese Objekte bereits ihrer Darstellung entsprechend als Sprite vorliegen, die gegebenenfalls noch mit Transformationen wie Skalierung an die jeweilige Situation angepasst, direkt dargestellt

werden können. Somit muss kein komplexes dreidimensionales Objekt auf einmal im Speicher gehalten werden, bei dem man sowieso nur die der Kamera zugewandten Seite sieht, sondern nur ein Spritesheet mit den möglichen Orientierungen und Bewegungsphasen des Objekts. Ebenso entfällt bei der Darstellung eine komplexe Berechnung der Überlagerung oder verdeckter Rückseiten der Objekte. Es wird nur die momentane Orientierung der Objekte dargestellt. Die Verdeckung von Objekten kann durch layerbasiertes Rendering gelöst werden:

1. Hintergrund rendern
2. Nicht Spieler Objekte rendern
3. Spieler Objekt rendern
4. Benutzeroberfläche rendern

Diese Besonderheiten vereinfachen die Berechnung im 2D-Darstellungsfall. Dadurch ist dieser Fall aber nicht so gut optimiert, wie die 3D-Anwendung, und es werden regelmäßig Grenzen ausgereizt.

Diese Problematik hat konkrete Auswirkungen auf den Ablauf von Videospielen. Hier das Beispiel des Speedrunnings:

Beim Speedrunning versucht der Spieler, so schnell wie möglich die Zielbedingung zu erfüllen. Das sind, je nach Spiel und Kategorie, das einsammeln aller Collectibles im Spiel, das Besiegen des Endgegners oder aber "nur" das Erreichen der Credits nach dem Spiel. Aus mehreren Gründen sind diese Spieler an einer guten Perfomanz des Spieles interessiert:

- Die Rekorde im selben Run (selbes Spiel, selbe Kategorie) sind oft nur Sekundenbruchteile schneller als die zweitschnellsten Ergebnisse.
- Zum Erreichen des Zieles dürfen, abhängig von der Kategorie, auch Spielfehler ausgenutzt werden, oder sogar durch Speichermanipulation und gezielten Over/Underflows das Spiel umgeschrieben werden. Hierfür sind oft pixelgenaue Positionierung in der Spielwelt und auf den Frame perfekte Benutzereingaben notwendig, um diese "Glitches" gezielt ausführen zu können.

Jeglicher Performance-Einbruch des Spieles erschwert die Jagd auf den nächsten Weltrekord. Die einfachste Lösung dieses Problems ist es, das Spiel auf immer stärkeren Rechnern zu starten. Fehlende Optimierung des Spiels wird durch mehr Rechenleistung ausgeglichen. Dies ist aber nicht immer möglich, vor allem bei Spielen, die nur auf Konsolen laufen. Hier muss der Speedrunner die fehlende Optimierung selber ausgleichen. Zum Beispiel:

- Betrachten des leeren Himmels: Es werden keine Objekte auf dem Bildschirm dargestellt - *Daggerfall, PC Speedrun*
- Das gezielte zerstören von bestimmten Gegnern: Es werden weniger Objekte auf dem Bildschirm dargestellt - *Mega Man 3, NES*

Dies sind auch erste Ansätze für die Sektion Identifizierte Bottlenecks.

3.2 Benchmarking

Um eine möglichst einfache Bedienung zu gewährleisten, soll der Benchmark vollautomatisiert erfolgen. Wichtigste Eigenschaft des Benchmarks ist die Vergleichbarkeit der Ergebnisse. Dazu wurde die *Renderzeit pro Frame bei X Objekten* als Kennzahl eingeführt. Wie bei allen empirischen Untersuchungen

muss diese den folgenden drei Kriterien genügen[1]:

Validität Dass sich die Kennzahl mit empirischen Messergebnissen deckt, ist garantiert, da sie empirisch zustande kommt. Es wurde die in Entwicklerkreisen übliche Größe *Renderzeit pro Frame in Sekunden* gewählt. Die Inverse dieser Größe, *Frames per Second*, ist zwar bei Nutzern und Marketing sehr beliebt, nicht jedoch bei technischen Entwicklern, unserer Zielgruppe. Ein arbiträres Punktesystem würde die Messwerte nur unnötig verschleiern und den Eigenschaften eines guten Benchmarks widersprechen.

Objektivität ist durch die Wohldefiniertheit der einzelnen Testverfahren gegeben (siehe 5.3). Die Implementierung in den einzelnen Frameworks soll möglichst gleich sein. Somit sind die verschiedenen Messwerte auf dem selben Computer nur auf die Effizienz der verwendeten Bibliothek zurückzuführen und nicht auf die Implementierung der Tests durch den verantwortlichen Entwickler.
Die BunnySuite soll speziell die Grafikleistung der verschiedenen Frameworks bewerten. Daher muss gewährleistet sein, dass das Testergebnis nicht maßgeblich von Berechnungen auf der CPU beeinflusst wird. Um die korrekte Implementierung der Tests zu überprüfen, kann eine Testreihe ohne graphische Ausgabe, also nur mit den CPU-Berechnungen der Update()-Phase ausgeführt werden. Um Vergleichbarkeit unter den Frameworks zu gewährleisten, liegen alle Benchmark-Implementierungen als binäre Executable-Datei vor, auch wenn manche Gameframeworks HTML5 als Zielplattform anbieten. Dabei würde das Messergebnis jedoch maßgeblich von der Effizienz des ausführenden Browsers beeinflusst, weshalb wir uns entschieden haben, auf die Zielplattform HTML5 vorerst zu verzichten.

Reliabilität Als Maßnahme für hohe Reliabilität werden alle Messungen standardmäßig jeweils zehn mal durchgeführt und anschließend der Durchschnitt gebildet. Dadurch dauert der gesamte Test zwar länger, aber es werden punktuelle Performance-Änderungen durch Hintergrundprozesse ausgeglichen. Die Anzahl der Wiederholungen kann optional per Parameter eingestellt werden.

Die standardmäßige Renderauflösung beträgt in allen Implementierungen 800 x 600 px, kann aber per Parameter angepasst werden.

[1] Siehe: `http://www.uni-leipzig.de/~emkf/Referate/G%C3%BCtekriterien%20PPT.pdf`

4 Konzept und Design

4.1 Allgemeines

Bei der Konzeption eines Benchmark gibt es grundsätzlich zwei verschiedene Herangehensweisen: Zum einen kann man eine repräsentative, in der Regel sehr komplexe Test-Szene rendern lassen, in der alle möglichen Schwierigkeiten und Bottlenecks vorkommen, die gemeinhin bei der Grafikprogrammierung auftauchen, und damit die GPU an die Grenze der Belastbarkeit zu bringen. Das ist der übliche Ansatz bei im Internet frei verfügbaren Benchmarks[1], die dazu dienen, die Fähigkeiten der eigenen Grafikkarte, also Hardware, zu testen. Zum anderen kann man viele einzelne, feingranulare Tests definieren, die auf einzelne Bottlenecks abzielen. Für das BunnySuite-Framework haben wir diese Herangehensweise gewählt, da das Framework explizit für den Vergleich von Grafik-Libraries konzipiert ist. Der Vorteil von feingranularen Tests ist, dass die Ergebnisse für die Entwickler sehr viel aussagekräftiger sind, weil sie ihnen präzise Hinweise geben, an welchen Stellen das eigene Framework hinter den anderen abfällt und wo eine Optimierung am meisten bringen würde. Außerdem vereinfacht dieser Ansatz die Entwicklung und Implementierung der Tests enorm. Ein Nachteil ist, dass die Tests teilweise artifizielle Anforderungen stellen, die in echten 2D-Spielen nicht vorkommen. Dieser Nachteil wird aber dadurch ausgeglichen, dass man die Tests – soweit sinnvoll – frei kombinieren kann. Außerdem ist die spätere Erweiterung des Frameworks um komplexere Tests durchaus möglich.

4.2 Identifizierte Bottlenecks

Da jede 2D-Grafik-Bibliothek ihre eigenen Stärken und Schwächen aufweist und für unterschiedliche Zwecke optimiert ist, ist es wichtig, differenzierte Tests durchzuführen, um gezielt mögliche Schwachstellen zu identifizieren, ohne dabei die Stärken zu ignorieren. Zu diesem Zweck wurden verschiedene "Bottlenecks" identifiziert, also häufige Schwachstellen, für die gezielt Tests entwickelt wurden. Diese sind unter anderem:

- Animieren der Objekte
- Objekte mit veränderlichen Texturen
- Vielzahl an gleichzeitig dargestellten Objekten mit unterschiedlichen Texturen
- Skalierung der Objekte
- Rotation der Objekte
- Objekte mit (halb-)transparenten Texturen
- Einfärbung von Texturen
- Darstellung von Primitiven wie Dreiecken.

Eine detaillierte Begründung findet sich in der Auflistung aller Testmodifier in Sektion 5.3.

[1] Ein Beispiel: `https://unigine.com/products/benchmarks/heaven/`

5 Umsetzung und Tests

5.1 Entwicklungsablauf

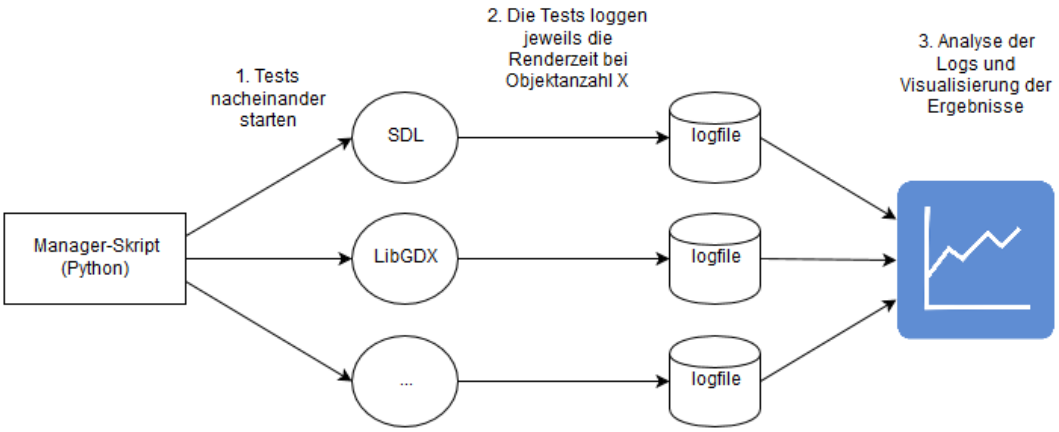
Im Laufe der Entwicklung wurde der Aufbau des Frameworks geändert. Der erste Entwurf sah die Modellierung von 3-4 großen Tests vor. So gab es einen Test, der unserem jetzigen Modifier *animation* entspricht und einen Test, der verschieden texturierte Bunnies an zufällige Positionen zeichnet (in der finalen Form eine Kombination aus den Modifiern *multitexture* und *random*).
Dies machte das Implementieren von neuen Tests schwierig und die Kombination von bestehenden Tests unmöglich.
Da wir nicht alle Szenarien, die für Spiele-Entwickler interessant sein könnten, vordefinieren konnten, wählten wir einen modularen Ansatz. Dieser schien uns insbesondere aus Anwendersicht vorteilhaft, da man sich somit schnell die für sich selbst interessanten Benchmarks definieren kann. Alle Abläufe wurden auf eine Kombination von elementaren Operationen runtergebrochen. Diese finden sich in den Testmodifiern wieder.

5.2 Ablauf der Tests

Der Ablauf eines Tests im *BunnySuite*-Frameworks ist in Abbildung 5.1 schematisch dargestellt. Das Framework besteht im Wesentlichen aus zwei Teilen:

1. Das **Manager-Skript** ist dafür zuständig, den spezifizierten Test für alle Frameworks nacheinander zu starten, anschließend die Logs zu analysieren und die Ergebnisse in einem Diagramm zu visualisieren.
2. Es gibt eine **ausführbare EXE-Datei** für jedes Framework. Diese Datei wird vom Manager-Skript gestartet und bekommt die Parameter für die Messung als Argumente über die Command-Line übergeben. Das Programm führt die spezifizierten Tests durch und loggt die Renderzeit pro Frame bei X Objekten.

Abbildung 5.1: Ablauf eines Tests



Der auszuführende Test wird durch folgende Parameter spezifiziert:

test_name Ein String aus den Einzusetzenden Testmodifiern (siehe 5.3). Die Namen der Modifier sind per Komma getrennt.
min_val Startwert für die Anzahl X der zu zeichnenden Objekte
max_val Endwert für X
step Schrittweite, um die X nach jeder Messung erhöht wird
xRes optional: Fensterbreite. Standardwert ist 800px
yRes optional: Fensterhöhe Standardwert ist 600px
repititions optional: Anzahl der Wiederholungen pro Messschritt, aus denen ein Durchschnittswert gebildet wird. Standardwert sind 10 Wiederholungen.

Ein Aufruf des Tests mit den Modifiern "animation" und "hd" mit doppelter Anzahl an Wiederholungsmessungen könnte also wie folgt aussehen:

App.exe animation,hd 0 10000 500 800 600 20

5.3 Beschreibung der Tests

In der folgenden Liste werden alle angebotenen Testmodifer genau definiert und ihre Auswahl für diesen Benchmark begründet:

animation

Beschreibung: Die Objekte werden mit zufälliger Geschwindigkeit in X- und Y-Richtung initialisiert. Mit jedem Frame wird die Position jedes Objektes mit seiner Geschwindigkeit neu berechnet. Die Y Geschwindigkeit wird zusätzlich durch "Schwerkraft" in jedem Frame beeinflusst. Wenn die Ränder des Frames erreicht werden, wird die Richtung umgekehrt. Dies führt zum Eindruck einer Hüpf-Animation für die Objekte. Die exakte Berechnungsformel kann dem Wiki im Bunnysuite-Github entnommen werden.[1]

Begründung: Dies ist der Testfall des ursprüngliches *BunnyMark*. Dieser Testmodifier simuliert die konstante Bewegung eines Objektes über den Bildschirm und stellt damit typische Situationen in Spielen nach.

alpha

Beschreibung: Zusätzlich zu den vorhandenen Texturen wird die Textur "wabbit_ghost" zum Texturenpool hinzugefügt. Diese Textur zeichnet sich durch ihre Halbtransparenz aus.

Begründung: Dieser Testmodifier zielt auf den korrekten und effizienten Umgang der Renderfunktionen des Frameworks mit Halbtransparenz ab. Hierfür müssen neue, korrekte Farbwerte aus der Textur und aller davon verdeckten Objekte berechnet werden.

bunnies

Beschreibung: Es werden explizit Bunnies gerendert.

[1] https://github.com/LiXiling/bunnysuite/wiki/Hasensprung-Routine

Begründung: Dieser Testmodifier erlaubt die Kombination von Primitiven mit texturierten Bunny-Objekten.

circles

Beschreibung: Anstelle von texturierten Bunnies wird ein Kreis mit einem Radius von 13 Pixeln und zufälliger Farbe gezeichnet.

Begründung: Mit diesem Testmodifier soll das Framework auf die korrekte und effiziente Berechnung und Darstellung von Kreisen getestet werden. Für die Darstellung ist die Annäherung von ganzahligen Pixeln an den Kreis besonders von Bedeutung.

colorchange

Beschreibung: Jedes Objekt, inklusive der texturierten Bunnies, ändert mit jedem Frame seine Färbung.

Intention: Mit diesem Testmodifier wird das Framework auf den Umgang mit schnellen Farbwechseln von Objekten, insbesondere Texturen, geprüft. Diese Funktion ist beispielsweise wichtig, um in Spielen einmal entworfene Objekte in verschiedenen Umgebungen wiederverwenden zu können.

hdtexture

Beschreibung: Zusätzlich zu den vorhandenen Texturen wird die Textur "wabbit_hd" zum Texturenpool hinzugefügt. Diese Textur ist sehr hoch aufgelöst und wird daher am Anfang auf die Höhe von 37px runterskaliert.

Intention: Mit diesem Testmodifier wird das Framework auf den effizienten Umgang mit sehr hochauflösenden Texturen geprüft. Es müssen große Dateien effizient gespeichert und gelesen werden, zusätzlich muss die GPU die große Menge an Pixelinformationen korrekt auswerten. Auch die optische Veränderung der Textur durch die jeweilige Implementierung der Skalierung im Framework ist zu beachten. Die Tests in Kapitel 6 zeigen nochmal die zusätzliche Belastung in diesem Fall.

lines

Beschreibung: Anstelle von texturierten Bunnies werden Linien mit den Endunkten (0,0) und (26,37) sowie mit zufälliger Farbe gezeichnet.

Intention: Mit diesem Modifier soll das Framework auf den korrekten und effizienten Umgang mit Linien geprüft werden. Zu beachten sind hier das Auftreten von Treppeneffekten bei diagonalen Linien (Aliasing). Linien können zum Beispiel für das schnelle Zeichnen von Plattformen oder die Darstellung von Sicht- oder Ziellinien nützlich sein.

multitexture

Beschreibung: Zusätzlich zu den vorhandenen Texturen werden die Texturen "wabbit_0", "wabbit_1" und "wabbit_2" zum Texturenpool hinzugefügt.

Intention: Dieser Modifier erhöht schnell die Anzahl der zur Verfügung stehenden Texturen im Texturenpool. Hierbei soll das Framework in seinem Speichermanagement getestet und der Realfall von Spielen mit einer großen Anzahl von verschiedenartigen Render-Objekten angenähert werden.

no_output

Beschreibung: Es findet keine Ausgabe an den Bildschirm statt. Sämtliche Zustandsberechnungen aus der Update-Phase werden entsprechend der restlichen Testparameter normal ausgeführt.

Intention: Dieser Modifier soll Entwicklern bei der Implementierung von den Tests in neuen Frameworks unterstützen. Damit kann überprüft werden, dass die Tests nicht zu CPU-intensiv in der Update-Phase berechnet werden. Dies ist notwending, da *BunnySuite* zur Überprüfung von Grafik-Frameworks gedacht ist.

points

Beschreibung: Anstelle von Bunnies werden Punkte in zufälliger Farbe gezeichnet.

Intention: Mit diesem Testmodifier soll das Framework auf das gezielte Malen einzelner Pixel geprüft werden. Dies ist beispielsweise für Partikeleffekte wichtig.

pulsation

Beschreibung: Mit jedem Frame wird die Skalierung der Objekte um ein "Wachstum" verändert (initial 0.1). Immer wenn die Skalierung >= 5 oder <= 0.2 ist, wird das Wachstum mit -1 multipliziert. Somit scheinen die Objekte zwischen den beiden Größenextremen hin und her zu pulsieren.

Intention: Mit diesem Testmodifier werden Objekte konstant skaliert, behalten also nicht ihre Größe bei. Dies soll das Grafikframework zusätzlich auslasten, da Objekte in ihren Dimensionen keine konstante Größe behalten und immer wieder neu für die Grafikausgabe berechnet werden müssen. Da die gleichen Skalierungsstufen immerwieder durchlaufen werden, können sich hier auch Caching-Effekte auswirken.

scaled

Beschreibung: Die Objekte werden am Anfang mit einem zufälligen Faktor zwischen 0.2 und 5.0 skaliert. Die X- und Y-Achse werden dabei unabhängig voneinander skaliert.

Intention: Mit diesem Testmodifier erscheinen Objekte mit stark unterschiedlichen Größen (und sogar Größenverhältnissen). Auch wird das Framework darauf geprüft, wie gut es Objekte nur in einer Dimension skalieren kann.

random

Beschreibung: Die Objekte werden am Anfang an eine zufällige Stelle gesetzt.

Intention: Dieser Testmodifier prüft über die Laufzeit des Tests hinweg zwei Merkmale: Am Anfang, wenn in der Regel wenige Objekte gezeichnet werden, findet kaum eine gegenseitige Verdeckung von Objekten statt. Im späteren Verlauf ist der gesamte Bildschirm mit Objekten gefüllt, frühe Objekte werden oft mehrmals übermalt. Auch hier soll eine reale Spielesituation angenähert werden, bei der der gesamte Bildschirm bemalt wird, und die Ausgabe nicht nur in einem sehr kleinem lokalen Bereich stattfindet.

rectangles

Beschreibung: Anstelle von texturierten Bunnies werden Rechtecke mit den Eckunkten (0,0) und (26,37) sowie mit zufälliger Farbe gezeichnet.

Intention: Mit diesem Testmodifer soll das Framework auf die korrekte und effiziente Darstellung von Rechtecken getestet werden. Diese Primitive können gut für Boxen oder Blöcke verwendet werden.

rotation

Beschreibung: Jedes Objekt wird pro Frame um 1 Grad im Uhrzeigersinn um seinen Mittelpunkt gedreht.

Intention: Mit diesem Testmodifer soll das Framework auf die korrekte und effiziente Berechnung und Darstellung der Rotation überprüft werden. Dies ist eine elementare Transformation, die oft verwendet wird.

teleport

Beschreibung: Die Bunnies werden in jedem Frame an neue zufällige Stellen gezeichnet.

Intention: Dieser Testmodifier hat eine ähnliche Intention wie der *random* Testmodifier. Durch die konstante Repositionierung der Objekte besteht nicht die Möglichkeit, Teile des letzten Frames wiederzuverwenden, es werden also Caching-Effekte außer Kraft gesetzt.

texts

Beschreibung: Anstelle von texturierten Bunnies wird der Text "Hello World!" gerendert.

Intention: Mit diesem Testmodifier auf das korrekte und effiziente Textrendering überprüft. Dies ist wichtig für die Darstellung von Text für den Spieler, bspw. im UI.

texturechange

Beschreibung: Mit jedem neuen Frame bekommt jeder Bunny eine neue zufällige Textur aus dem Texturenpool zugewiesen.

Intention: Mit diesem Testmodifier wird das Speichersystem des Frameworks und die Optimierung der Renderfunktionen auf andauernd wechselnde Texturen überprüft. Ein realistisches Szenario für schnellen Texturwechsel sind Animationsphasen, hier werden oft einfach eine Reihe von leicht verschiedenen Texturen durchgespielt, um den Eindruck einer Bewegung zu erwecken.

thin

Beschreibung: Zusätzlich zu den vorhandenen Texturen wird die Textur "wabbit_y" zum Texturenpool hinzugefügt, die nicht einer Rechteck-Annäherung entspricht und damit viel Voll-Transparenz aufweist.

Intention: Diese Textur ist nur schlecht durch ein Rechteck angenähert. In einer effizienten Implementierung des Renderings sollten die volltransparenten Bereiche der Textur übersprungen werden. Diese Effizienz wird hierbei überprüft.

tinted

Beschreibung: Jedes Objekt, insbesondere texturierte Bunnies, werden mit zufälliger Färbung erstellt.

Intention: Mit diesem Testmodifier wird überprüft, ob das Framework korrekt mit der Einfärbung von Texturen umgehen kann. Auch findet hier eine zusätzliche Rechenbelastung statt, da die Textur selber nicht einfach eingelesen werden kann, sondern eine korrekte neue Farbe effizient berechnet werden muss.

triangles

Beschreibung: Anstelle von texturierten Bunnies werden Dreiecke mit den Eckunkten (0,37), (26,37), und (13,0) sowie mit zufälliger Farbe gezeichnet.

Intention: Dreiecke sind in der Spielegrafik von besonderer Bedeutung. 3D Objekte und komplizierte 2D-Formen werden durch von Dreiecken beschriebenen Polygonen angenähert. Die Effizienz im Umgang mit Dreiecken wird von diesem Testmodifier geprüft.

5.4 Implementierungsdetails bei den einzelnen Frameworks

5.4.1 Kha

Kha ist ein open-source Framework, entwickelt von Robert Konrad. Es basiert auf Haxe und lässt sich auf eine Vielzahl von Plattformen und Engines portieren. Für den BunnySuite wurden die Tests in der eigenen IDE *Kode* entwickelt und anschließend für die Zielplattform Windows in Visual Studio kompiliert. Mit Kha wurden unter anderem *Spiral Ride* oder *Dungeons Deep* realisiert.

5.4.2 SDL

SDL (Single DirectMedia Layer) ist eine multi-plattform Bibliothek, die low-level Zugriff auf die Hardware (Audio, Grafik, Eingabegeräte) über OpenGL und Direct3D anbietet. SDL ist in C geschrieben und von Haus aus mit C++ kompatibel. Es stehen aber auch Portierungen für C# oder Python zur Verfügung. Für das BunnySuite-Projekt wurde SDL 2 mit C++ verwendet. Beispiele für Spiele, die SDL einsetzen, sind *Angry Birds*, *Dont't Starve* und auf der Open Source Seite *The Battle for Wesnoth*. Die offiziell unterstützten Plattformen sind Android, Linux, Mac OS und Microsoft Windows.
SDL ist im Vergleich zu den anderen Bibliotheken sehr low-level und stellt nur sehr grundlegende Funktionalitäten bereit. Um die Primitiven zu implementieren, wurde zusätzlich die Bibliothek *SDL2_gfx*[2] von Andreas Schiffler verwendet.

[2] http://www.ferzkopp.net/Software/SDL2_gfx/Docs/html/index.html

5.4.3 XNA/Monogame

XNA (XNA's Not Acronymed) ist ein von Microsoft entwickeltes Set von Tools zur Game Entwicklung. XNA basiert auf dem .NET Framework und läuft nativ auf Windows, Windows Phone und XBox360. Es war gedacht als einfache Alternative zu DirectX. XNA wird von Microsoft selber nicht mehr weiterentwickelt. Es exisitiert aber eine OpenSource Portierung namens Monogame, welche weiterhin aktiv gepflegt wird. Monogame läuft auch auf den Plattformen iOS, Mac OS, Android, Linux, Playstation 4 und PSVita. Beispiele für Spiele, die auf Monogame basieren, sind *Fez* oder *Transistor*.

5.4.4 LibGDX

LibGDX ist ein auf Java basierendes quelloffenes Framework. Es wurde entwickelt, da der Hauptentwickler beim programmieren von Android-Spielen schnell auf dem Desktop geschriebene Code-Änderungen testen wollte. Einer der Vorteile von LibGDX ist die Abstraktion der Unterschiede von Desktop-Plattformen wie Linux/Windows und Mobile-Plattformen wie Android/iOS. Beispiele für Spiele, die LibGDX einsetzen sind *Animal Memory for children* oder *Kid on Crack*.

5.5 User Interface

5.5.1 GUI

Um das Starten von Benchmarks und die Auswahl der Testmodifier möglichst einfach zu gestalten, wurde ein Graphical User Interface entwickelt. Abbildung 5.2 zeigt einen Screenshot. Man kann die zu testenden Bibliotheken und die Testmodifier auswählen. Anschließend legt man für die Anzahl der zu zeichnenden Objekte den Startwert, den Endwert, die Schrittweite und die Anzahl der Wiederholungen pro Schritt fest. Außerdem kann die zu verwendende Bildschirmauflösung eingestellt werden. Über die Funktion "Presets" können Konfigurationen zur späteren Wiederverwendung gespeichert werden, dazu einfach einen Namen eingeben und auf "Save" klicken. Nach einem Klick auf "Run" werden die Tests gestartet und durchgeführt. Wenn die Tests abgeschlossen sind, werden zwei Diagramme angezeigt und zur weiteren Verwendung im Unterordner "results" gespeichert:

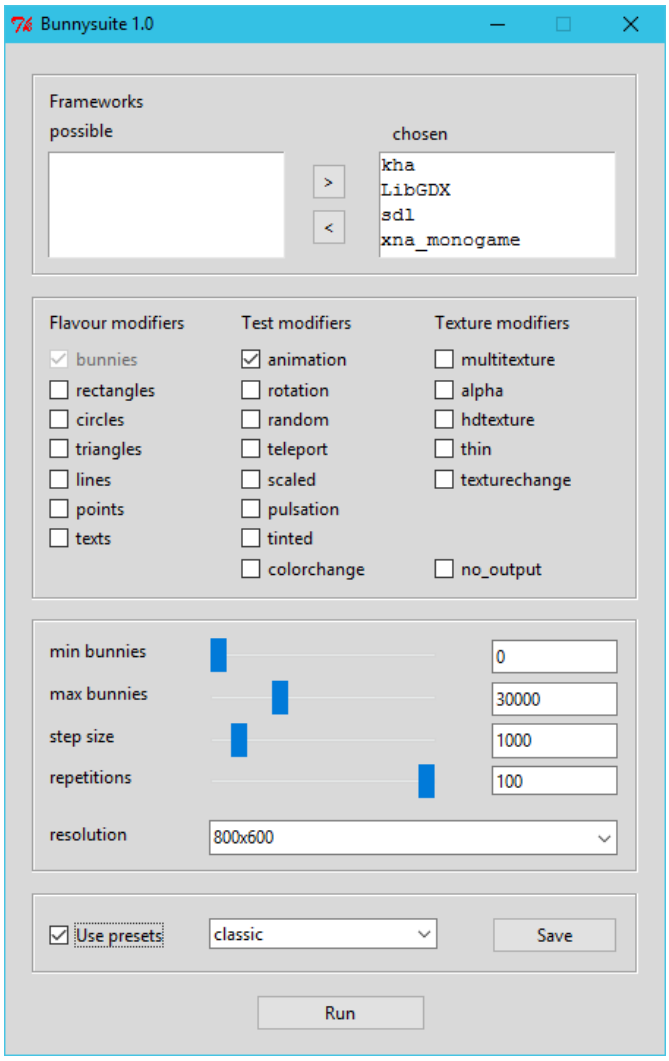
1. Ein Balkendiagramm, dass für jede Bibliothek angibt, ab welcher Anzahl von Objekten die Framerate spürbar unter 60 fps absinkt (Abbildung 5.3a).
2. Ein Liniendiagramm mit den Renderzeiten aller Bibliotheken in Abhängigkeit von der Anzahl der Objekte (Abbildung 5.3b).

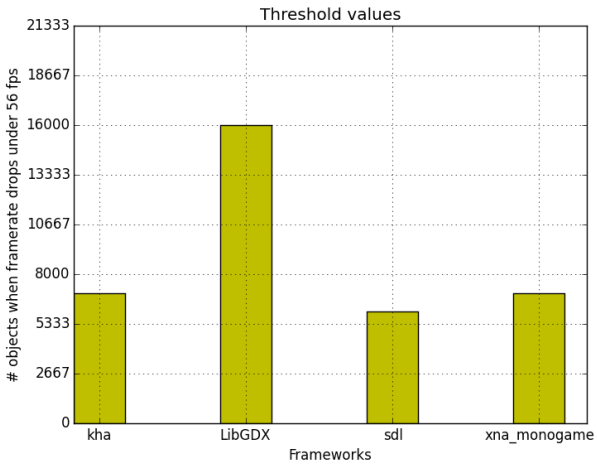
Außerdem wird automatisch ein Report generiert, der die Messwerte für alle Frameworks tabellarisch aufführt und zudem die Testparameter angibt.

5.5.2 Tests direkt starten

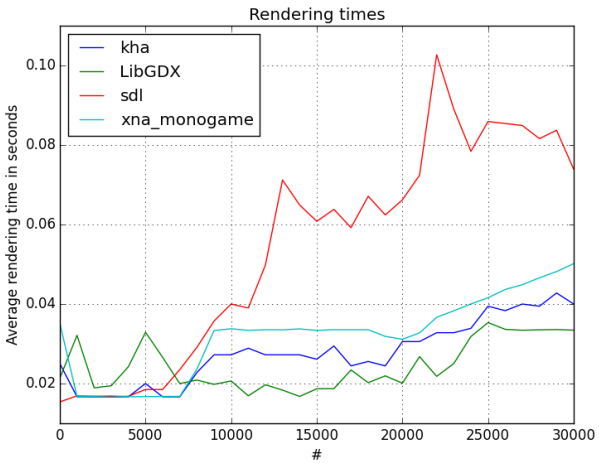
Tests können auch ohne GUI direkt gestartet werden. Dazu müssen die zu testenden Frameworks und die gewünschten Werte für die Parameter in die Datei *run_test.py* hineingeschrieben werden. Anschließend startet man dieses Skript. Der restliche Ablauf (Durchführung der Tests und Ausgabe des Diagramms) ist genauso wie bei der GUI, insbesondere werden auch die beiden Diagramme und der Report im Unterordner "results" erzeugt.

Abbildung 5.2: Screenshot der BunnySuite-GUI





(a) Unterschreiten von 56 FPS.



(b) Renderzeit pro Anzahl Objekte.

Abbildung 5.3: Die generierten Diagramme.

6 Ergebnisse der Messungen

In diesem Kapitel stellen wir ein paar interessante Ergebnisse von Messungen vor, die wir mit den Frameworks durchgeführt haben.
Dazu nutzen wir die beiden Computersysteme, deren Eckdaten in Tabelle 6.1 zu finden sind. Bei System 1 handelt es sich um ein günstiges Notebook, während System 2 ein typischer Gamer-PC ist.

6.1 XNA Forced VSync?

Diese Messung fand auf System 2 mit folgenden Parametern statt:

modifiers: bunnies, random, pulsation
frameworks: kha, LibGDX, sdl, xna_monogame
minBunnies: 0
maxBunnies: 100000
stepSize: 1000
repetitions: 10
resolution: 1280x720

Es wurden somit alle 10 Frames 1000 neue Bunnies mit Standardtextur an eine zufällige Position gezeichnet. Zusätzlich wurden alle Bunnies mit jedem Frame neu skaliert.
Diagramm 6.1 zeigt die Ergebnisse. Man sieht, dass XNA bei 62000 Bunnies in der Renderzeit plötzlich einen Sprung von 0,016s = 60fps zu 0,033s = 30fps vollzieht. SDL und LibGDX zeigen hier einen linearen Anstieg, so wie es erwartet wird. Kha macht um die 72000 Bunnies auch einen Sprung von 0,016s zu 0,025s = 40fps, und oszilliert anschließend um diesen Wert. Hier könnte Textur-Caching eine Rolle spielen.
Das abnormale Verhalten von XNA ist hier von besonderem Interesse. Mit diesem Testsetup lässt sich das Verhalten von XNA zuverlässig reproduzieren, tritt aber in anderen Setups nicht mehr auf. Stattdessen kommt es auch hier zu einem linearen Anstieg der Renderzeit (siehe Diagramm 6.2). Da die Renderzeiten hier genau auf die übliche Bildwiederholungsrate von 60Hz passt, liegt die Vermutung nahe, dass es sich hier um aktiviertes VSync handelt, um Tearing zu verhindern. Dies tritt auf, wenn eine Ausgabe auf den Bildschirm erfolgen soll, die genau mit der Refresh-Phase VBlank des Monitors zusammenfällt. Um dies zu verhindern, wird die Ausgabe mit diesen Phasen synchronisiert. Diese Option ist aber bei der Implementierung der Tests im Framework explizit deaktiviert worden. Ebenso spricht das normale Verhalten in anderen Tests gegen diese These.

	System 1	System 2
Modellname	Acer Aspire E1-531	Custom-PC
CPU	Intel Pentium 2020M (2,4 GHz)	Intel Xeon E3-1231 v3 @ 4 x 3.4 GHz
GPU	Intel HD Graphics	NVidia Geforce GTX 780
RAM	4096 MB	16384 MB

Tabelle 6.1: Spezifikationen der beiden Testrechner

Abbildung 6.1: Ergebnis der XNA Forced VSync Messung

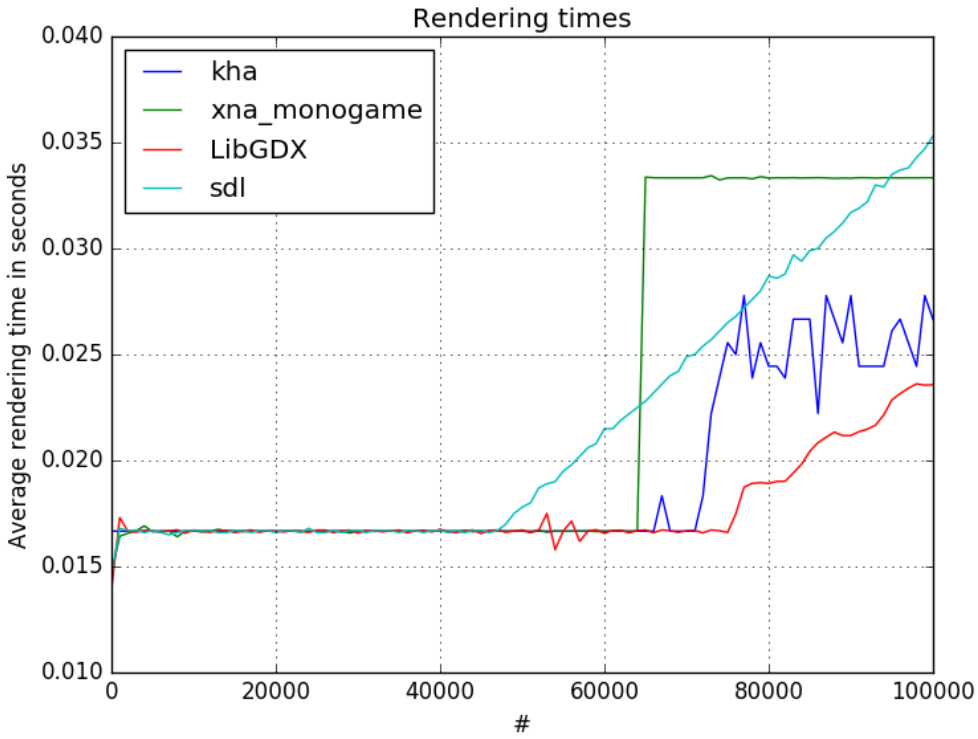


Abbildung 6.2: Ergebnis Diagramm: Normales XNA Verhalten

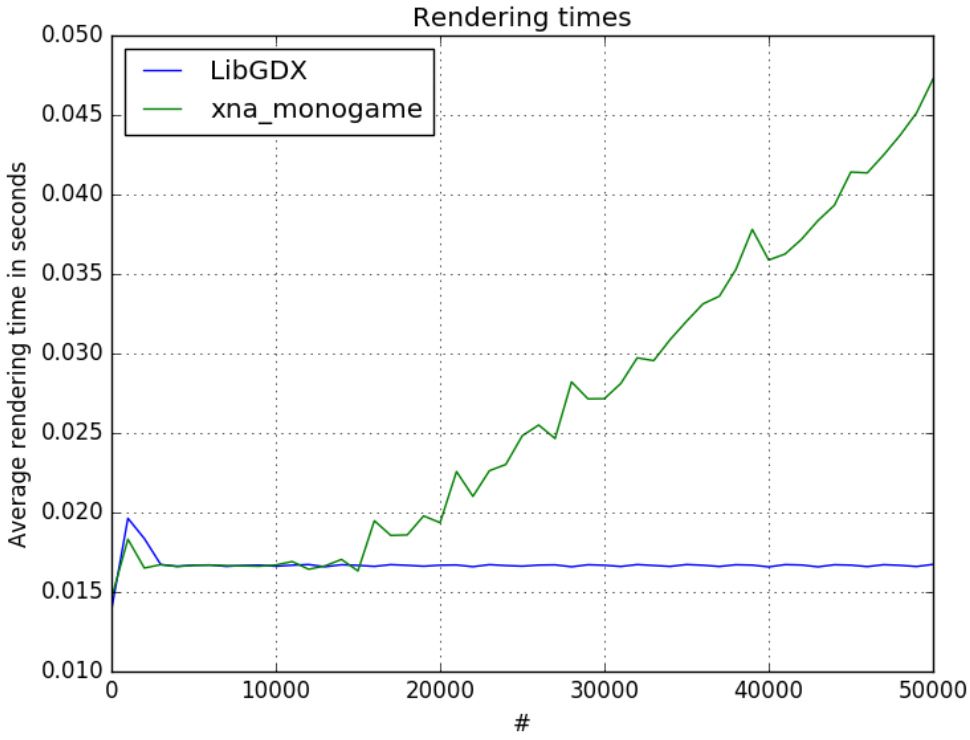
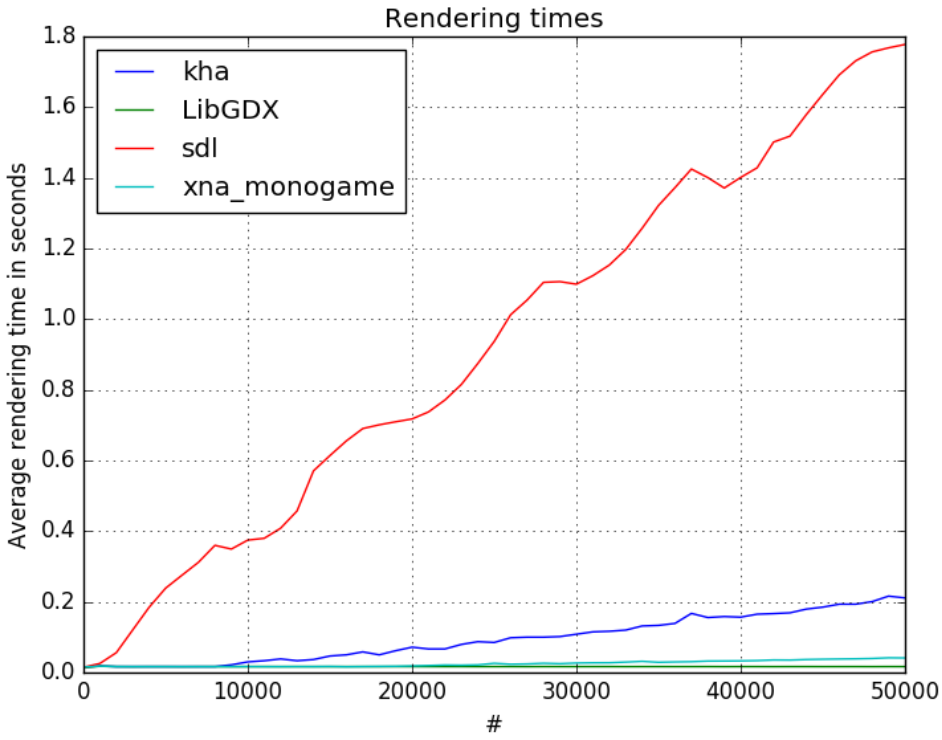


Abbildung 6.3: Ergebniss Diagramm der Primitiven Messung

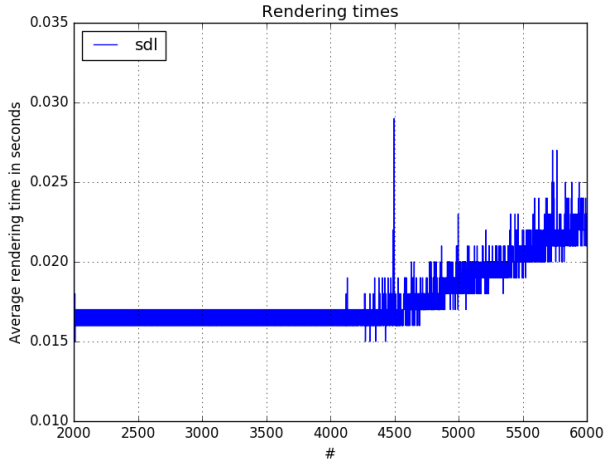


6.2 SDL Primitive

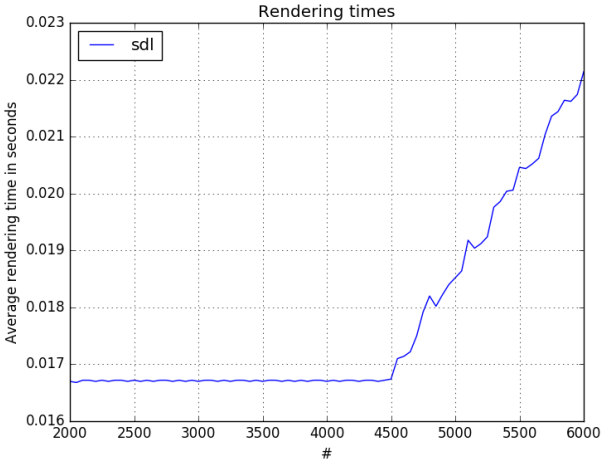
Diese Messung fand auf System 2 mit folgenden Parametern statt:

modifiers: rectangles, circles, triangles, lines, points, rotation, random, pulsation
frameworks: kha, LibGDX, sdl, xna_monogame
minBunnies: 0
maxBunnies: 50000
stepSize: 1000
repetitions: 10
resolution: 1280x720

Es wurden somit alle 10 Frames 1000 zufällige Objekte der Primitive Dreieck, Rechteck, Kreis, Linie und Punkt an eine zufällige Screen-Position hinzugefügt. Jedes Objekt ändert mit jedem Frame seine Skalierung. Die Ergebnisse werden in Diagramm 6.3 dargestellt. Bei der Darstellung und Skalierung von Primitiven hat SDL massive Probleme. Während die anderen Frameworks weit unter 1s Renderzeit bleiben, erreicht SDL bei 26000 Objekten eine Renderzeit von 1s, und erreicht gegen Ende 1,8s Renderzeit. Das ist 9 mal so lange wie das nächstlangsame Framework. SDL erreicht nie den Idealwert von 0,016s = 60fps Renderzeit und schon bei 2000 Objekten wird eine Renderzeit von 0,1s = 10fps erreicht. Die Ergebnisse zeigen, dass *SDL2_gfx*, das für das Rendering von Primitiven eingesetzt wurde, noch nicht besonders gut optimiert ist.



(a) Schrittweite und Repetitions jeweils 1



(b) Schrittweite und Repetitions jeweils 50

Abbildung 6.4: Vergleich von zwei Messungen derselben Dauer, aber mit unterschiedlichen Parametern

6.3 Die Parameter Repetitions und Schrittweite

Diese Messung fand auf System 1 mit folgenden Parametern statt:

modifiers: bunnies, random
frameworks: sdl
minBunnies: 2000
maxBunnies: 6000
stepSize: 1 bzw. 50
repetitions: 1 bzw. 50
resolution: 800x600

Die Renderzeit für eine bestimmte Framerate wird mehrmals gemessen und anschließend wird der Durchschnitt berechnet. Wie oft die Messung pro Framerate wiederholt wird, wird vom Parameter *Repetitions* festgelegt. Je höher *Repetitions* gesetzt wird, desto länger dauert die Messung. Der andere Parameter, der die Dauer der Messung maßgeblich beeinflusst, ist die Schrittweite *stepSize*, die festlegt, wie weit die Messpunkte auseinander sind. Erhöht man die Schrittweite um denselben Faktor wie *Repetitions*, dann dauert die Messung ungefähr gleich lang. An diese Feststellung schließt sich die Frage an, ob es sinnvoller ist, mit kleiner Schrittweite und wenigen Repetitions zu messen oder mit großer Schrittweite und vielen Repetitions.
Die Diagramme 6.4a und 6.4b zeigen den Unterschied. Links sieht man, dass bei zu wenigen Repetitions die Varianz der Messwerte sehr groß wird. Da die Messwerte wegen der geringen Schrittweite auch nahe aneinander liegen, ist das Diagramm kaum noch lesbar. Rechts hingegen erkennt man gut den linearen Anstieg der Framerate ab ungefähr 4500 Bunnies. Der Vergleich zeigt, dass es für aussagekräftige Diagramme sinnvoller ist, mit vielen Repetitions zu testen, als mit einer sehr feinen Schrittweite. Letzteres ist hingegen sinnvoll, wenn man einen möglichst genauen Treshold-Wert für die Anzahl der Objekte ermitteln will, bei dem die Framerate einen bestimmten Wert unterschreitet. Aber auch hierbei sollte die Anzahl der Repetitions nicht zu klein gewählt werden, sonst wird der ermittelte Wert zu hoch (aufgrund von Ausreißern nach unten bei der Renderzeit).

7 Zusammenfassung und Ausblick

Im Rahmen des Serious Games Praktikums wurde *BunnySuite* entwickelt, ein Benchmark-Framework für 2D-Grafik-Libraries. Im Vergleich zu BunnyMark sind die Messungen von BunnySuite vollautomatisch und ohne Benutzerinteraktion durchführbar, außerdem sind die Testergebnisse detaillierter und werden dem Benutzer systematischer präsentiert. Es wurde eine Vielzahl von Testfällen (Rotation, Skalierung, Primitive, etc.) identifiziert, welche im Framework als Test-Modifier frei kombinierbar sind. Für die vier Bibliotheken Kha, LibGDX, SDL und XNA/Monogame wurden Implementierungen entwickelt. Bei der Entwicklung wurde vor allem auf eine hohe Vergleichbarkeit geachtet. Deshalb wurde der Effekt von allen Test-Modifiern genau definiert.
In dieser Ausarbeitung wurden außerdem einige Tests analysiert, die mit BunnySuite durchgeführt wurden. Dabei hat sich gezeigt, dass BunnySuite-Tests aufgrund ihrer Modularität gut dazu geeignet sind, Schwachstellen von Bibliotheken gezielt offenzulegen.
Als Future Work kann BunnySuite noch um weitere Frameworks erweitert werden. Insbesondere Frameworks für 2D-Grafik im Browser mit der Zielplattform HTML5 sollten auch im Browser ausgeführt werden und verglichen werden können. Hierbei besteht das Problem, dass man aus einem Browser heraus keine Logfiles anlegen kann, da ein Zugriff auf das lokale Dateisystem aus Sicherheitsgründen nicht möglich ist. Das könnte mit einem Automatisierungs-Tool wie Selenium[1] gelöst werden.
Außerdem können die Implementierungen für die vier Bibliotheken, insbesondere die für SDL, noch auf Optimierungspotential untersucht werden.

[1] `http://www.seleniumhq.org/`