Beschreibung der Applikation *Star Flowers*

Frederik Brudy[1], Janko Hofmann[1]

[1]University of Munich (LMU)
Amalienstraße 17, 80333 München
12. April 2013

Gruppe: Star Flowers

brudy@cip.ifi.lmu.de, hofmannj@cip.ifi.lmu.de

1 Was ist es?

Star Flowers ist eine interaktive Kunstsimulation. Ein Nutzer kann mit einem, anfangs leeren, Display interagieren und so seine eigene Blumenwiese säen. Wie im echten Leben, so auch in der Kunst: Solange er sich um seine Blumen kümmert, solange geht es ihnen gut. Sobald er die Blumen sich selbst überlasst und vom Bild wegtritt, verwelken die Blumen und gehen ein. Final ist das Display in seinem Ausgangszustand vorzufinden und bereit für den nächsten Gärtner.

Die Applikation läuft auf einem öffentlichen Display, mit angeschlossenen Microsoft Kinect Sensoren. Die Interaktion erfolgt durch Gesten und auf Bewegungen des Nutzers hin.
Sobald die Sensoren einen vorbeigehenden Passanten erfassen, wird versucht dessen Aufmerksamkeit auf das Display zu locken: Der Schatten der Person wird auf dem Display angezeigt und im Zentrum des Körpers wird ein Partikelsystem erzeugt, welches große und schnelle Partikel erzeugt um zusätzlich im peripheren Blickwinkel aufzufallen.
Wenn der Nutzer sich nun zum Display wendet, erscheinen an seinen Händen Partikelsysteme, mit Hilfe derer er am unteren Displayrand eine Blumenwiese kreieren kann: Wenn er sich nach unten beugt, wächst an der Handposition eine Blume. So kann jeder seine ganz persönliche Anzeige gestalten. Sobald der Nutzer das Display verlässt, gehen langsam die Pflanzen wieder ein, bis das Display leer ist und bereit für den nächsten Passanten.

2 Für wen ist es?

Die Simulation ist konzipiert für Passanten, die am Public Display an der Frontfassade der Amalienstr. 17 vorbeigehen. Dieses ist im ausgeschalteten Zustand relativ unscheinbar, wird aber durch die Simulation interaktiv und reagiert auf Personen vor der Fassade.
Das gesamte Projekt ist so konzipiert, dass es “im Vorbeigehen” funktioniert, also Personen ermöglicht, für kurze Zeit damit zu interagieren, es zu erforschen und sich daran zu erfreuen, was der Aufmerksamkeitsspanne für vorbeigehende Passanten entspricht. Dies wird unterstützt durch eine schnelle Reset-Phase, welche die Simulation wieder in den Ausgangszustand versetzt, um die nächsten Passanten zu unterhalten.

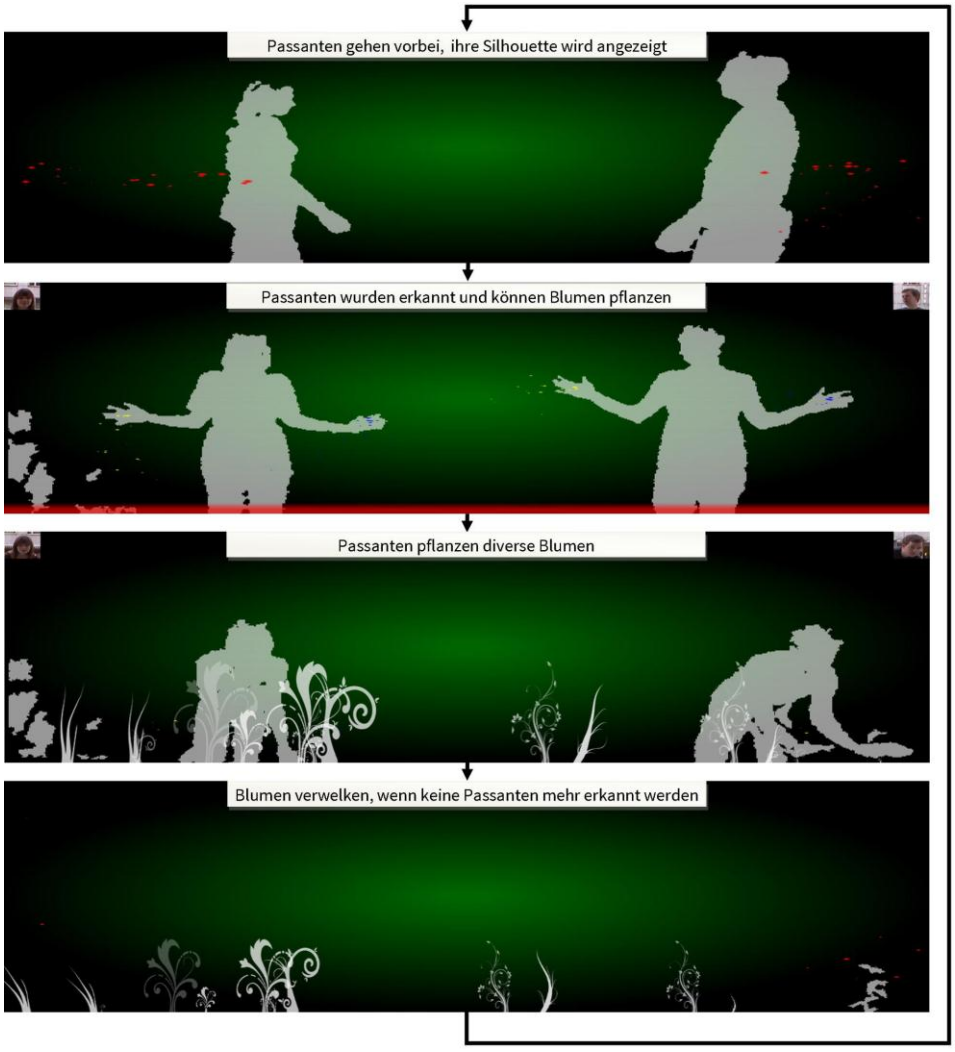
3 Wie funktioniert es?

Sobald eine Person am Public Display vorbeigeht, wird deren Silhouette (erfasst aus dem Depth Stream der Kinect) auf dem Monitoren angezeigt und in deren Mitte rote Partikel ausgesendet. Durch die Bewegung wird die Aufmerksamkeit vorbeigehender Personen erregt. Wenn diese stehenbleiben und deren Skelett von der Kinect erfasst wird, wechselt das Programm in einen weiteren Modus, der zusätzlich das Gesicht der gerade erkannten Person in der oberen Bildschirmecke anzeigt und ausgehend von deren Händen Partikel (“virtuelle Samen”) erzeugt, mit der Blumen gepflanzt werden können. Zusätzlich blinkt am unteren Bildschirmrand eine Leiste rot auf, die den Nutzern anzeigt, dass sie an dieser Stelle mit der Simulation interagieren können.
Die Simulation kann von zwei Personen gleichzeitig benutzt werden, wobei jeweils eine Person von einem Kinect-Sensor erfasst wird. Sie funktioniert aber auch für eine Person allein. Die Daten von beiden Sensoren werden getrennt abgefragt, beeinflussen aber gemeinsam die Simulation. So kann, wenn eine Person bereits Pflanzen sät, eine zweite hinzutreten und, nachdem sie im Bruchteil einer Sekunde erkannt wurde, ebenfalls mitmachen, oder die andere Person ablösen, ohne dass dies die Simulation negativ beeinflusst.
In der grafischen Ausgabe der Simulation gibt es keine statischen Objekte auf den Bildschirmen. So wird das Einbrennen auf den Displays zuverlässig verhindert:

- Die Silhouetten der Person folgen ihren Bewegungen
- Die Partikelzentren folgen den Personen und die einzelnen Partikel sind ständig in Bewegung
- Die Bilder der Gesichter folgen ebenfalls der Bewegung der Personen

- Die Pflanzen wiegen sich im Wind und bewegen sich zusätzlich um eine zufällige Distanz horizontal am unteren Bildschirmrand
- Der Hintergrund besteht aus einem Farbverlauf mit schwachem Kontrast, der nicht sichtbar einbrennen kann

Zustände der Simulation:



4 Besonderheiten / Funktionsweise / Beispiele

4.1 Verschiedene Prozesse und Threads des Systems

Die Anwendung besteht aus zwei Programmen: Die Hauptanwendung mit dem grafischen Interface und einer Konsolenapplikation, die Daten von einem Kinect-Sensor empfängt und der Hauptanwendung bereitstellt.
Die Hauptanwendung startet dynamisch so viele Prozesse der Konsolenanwendung, wie Kinect-Sensoren angeschlossen sind.
Sowohl Hauptanwendung wie auch Konsolenapplikation laufen endlos lange, bis sie geschlossen werden.
In der Hauptapplikation kümmert sich ein Thread um das Empfangen der Daten der angeschlossenen Sensoren. Ein weiterer Thread sorgt für die Animation der Image-Sprites (der Pflanzen) und ein dritter für die Berechnung und Darstellung der Partikelsimulation. Generell werden alle Operationen zum Zeichnen dem UI-Thread (per Dispatcher) übergeben und Berechnungen in separaten Threads ausgeführt. Dies stellt sicher, dass z.B. die Partikelsimulation und die Sprite-Animationen der Pflanzen durchgängig flüssig dargestellt werden, selbst wenn die Bewegungen zweier Personen von zwei Kinect-Sensoren rechenintensiv verarbeitet werden müssen.
In der Konsolenanwendung werden im Event-Handler `SensorAllFramesReady` kontinuierlich alle drei Streams der angeschlossenen Kinect ausgelesen und in einen mit der Hauptanwendung gemeinsam genutzten Speicherbereich geschrieben.

4.1.1 Wie ist der Durchlauf des Programms beim Start?

Der Einstieg in die Startroutinen der WPF Applikation erfolgt in der Methode `private void WindowLoaded(object sender, RoutedEventArgs e)`. Hier werden nicht nur die Kinect Sensoren initalisiert, sondern auch Sprite Bilder geladen, Image Container und Brushes vorbereitet und das Partikelsystem initialisiert.
Nach abgeschlossener Initialisierung wird der für das Empfangen der Daten von den Kinect-Sensoren zuständige Thread gestartet. Dort werden pro angeschlossenem Sensor je eine `MemoryMappedFile` für `ColorStream`, `DepthStream` und `SkeletonStream` angelegt und initialisiert und darauffolgend für jeden angeschlossenen Sensor ein Prozess der Konsolenapplikation gestartet. Den Prozessen wird als Kommandozeilenparameter die UniqueID des jeweiligen Kinect-Sensors sowie ein Prozess-ID übergeben, mit der die MemoryMappedFiles identifiziert werden.

Anschließend wird in einer Endlosschleife sequentiell von beiden Sensoren alle drei Streams aus den entsprechenden `MemoryMappedFiles` ausgelesen und einer Methode übergeben, die sie weiterverarbeitet.

4.1.2 Wie ist der Ablauf eines normalen Programmzyklus?

In jedem Prozess der Konsolenapplikation wird vom Kinect SDK automatisch die Callback-Methode `SensorAllFramesReady` aufgerufen, in der aus dem `SkeletonStream` alle Skeletons ausgelesen werden und für jedes überprüft wird, ob es den Status “tracked” hat. Nur diese werden in einem Array gespeichert. In der Weiterverarbeitung wird jeweils nur noch das erste getrackte Skeleton berücksichtigt. Dieses wird in ein `byte`-Array serialisiert und in die entsprechende `MemoryMappedFile` geschrieben. Wurde keine getracktes Skeleton gefunden, wird die `MemoryMappedFile` mit Nullen befüllt. Analog wird anschließend mit dem `ColorStream` und dem `DepthStream` verfahren, die bereits als Arrays vom Typ `byte` bzw. `DepthImagePixel` vorliegen. Jeder Zugriff auf eine `MemoryMappedFile` ist durch eine `Mutex` geschützt, um Nebenläufigkeits-Probleme zu vermeiden. Nach der Übertragung in die `MemoryMappedFile` legt sich der Thread 25ms schlafen und beginnt anschließend von vorn. Auf eine exakte zeitliche Synchronisierung mit der Framerate der Hauptanwendung haben wir bewusst verzichtet, da es keine Rolle spielt, ob die neuen Daten der Sensoren einen Frame früher oder später angezeigt werden, da das bei 33 Frames pro Sekunde schlicht nicht auffällt.

In der Hauptapplikation wird in einem eigenen Thread in einer Endlosschleife nacheinander die Daten aller Sensoren aus den entsprechenden `MemoryMappedFiles` ausgelesen. Das Skeleton wird wieder deserialisiert.
Anschließend werden Arrays mit der Länge der Anzahl angeschlossener Kinect-Sensoren mit jeweils deren `Skeletons`, `ColorStreams` und `DepthStreams` an eine Methode übergeben, die sie weiterverarbeitet.

In dem Thread welcher durch `doPlantDrawing()` bezeichnet ist wird alle 30ms die grafische Ausgabe geregelt: Im Kontext des UI-Threads wird nacheinander der Fade der triggerArea kontrolliert, anschließend bereits wartende Blumen gepflanzt und bereits wachsende Blumen vergrößert, bzw. deren Animation durchgeführt. Der Ablauf findet in `private void doActualPlantDrawing()` statt und ist im Detail in Kapitel 4.4 beschrieben. Anschließend erfolgt das generieren der Partikelsysteme, welches in Kapitel 4.5 genauer beschrieben wird und in `private void doFrameCalculation()` erfolgt.

Da die grafische Ausgabe in einem separaten Thread läuft und das meiste Rendering auf der GPU erfolgt ist sie mit 33 FPS besonders performant (limitiert durch unseren Thread. Wir erreichten in Tests bis zu 65FPS).

4.2 Aufmerksamkeit erregen: Körperzentrum berechnen und TriggerArea

Wenn Personen seitlich am Public Display vorbeigehen, können die Kinect-Sensoren deren Skelette noch nicht erkennen, allerdings können deren Tiefensensoren die Personen aufnehmen.
Daher muss in dieser Phase die Aufmerksamkeit der Passanten geweckt werden, damit sie sich zum Display drehen und die Kinect-Sensoren ihr Skelett erfassen können.
Wenn kein Skelett erkannt wurde, wird in jedem Frame von jedem Kinect-Sensor das Zentrum der Form berechnet, dass die Tiefensensoren in einem bestimmten Abstand erfassen, die also mutmaßlich zu einer vorbeigehenden Person gehört. An diesem Punkt werden dann Partikel generiert.

Für jeden Pixel des DepthStreams, der zwischen 0.5 und 2.5m entfernt ist:

```
/* wenn kein Skeleton erkannt, alle Punkte aus dem Depth-Shape
addieren, um später Mittelpunkt zu berechnen */
playerXPoints += colorInDepthX;
playerYPoints += colorInDepthY;
playerPointCount++;
```

wenn über alle Pixel iteriert wurde:

```
// wenn überhaupt etwas im Zielbereich getrackt wurde
if (playerPointCount > 0) {
	// Mittelpunkt aus Depth Shape berechnen
	playerCenterPoint = new System.Windows.Point (
		(double)playerXPoints / playerPointCount,
		(double)playerYPoints / playerPointCount );
}
```

Am `playerCenterPoint` werden dann fortlaufend rote Partikel erzeugt, die die Silhouette der Person überlagern. Sobald sich der Passant dann zum Display gewendet hat und sein Skelett erkannt wurde ist es seine Aufgabe mit seinen Händen am unteren Bildschirmrand Blumen zu “säen”. Damit ein neuer Nutzer dies auch weiß wird am unteren Bildschirmrand ein roter Farbverlauf sanft ein- und ausgeblendet.

Die Initialisierung erfolgt bei Programmstart:

```
TriggerArea.Fill =
ColorsAndBrushes.getBrushFromColorLinearGradient(Colors.Red);
TriggerArea.Opacity = 0.0; //zu Beginn verstecken
TriggerArea.Width = this.Width; //maximale Breite
TriggerArea.Margin = new Thickness(0, this.Height -
TriggerArea.Height, 0, 0); //am unteren Bildschirmrand
positionieren
```

Innerhalb eines Threads wird alle 30ms das eigentliche ein- und ausfaden durch Änderung der Opacity geregelt: In `private void fadeTriggerArea()`.

```
if (TriggerArea.Opacity < 0.1) {
      triggerDiff = 0.05; //einfaden
}
else if (TriggerArea.Opacity > 0.9) {
      triggerDiff = -0.05; //ausfadenb
}
TriggerArea.Opacity +=triggerDiff;
```

4.3 Know-Who-Is-Playing – Gesichtserkennung

Damit der Nutzer weiß, welche Displayhälfte er kontrolliert und welche Person im Moment bei mehreren Nutzern erkannt wurde, wird das Gesicht der Person passend aus dem `ColorStream` ausgeschnitten und in der oberen Ecke angezeigt. Die Erkennung erfolgt auf Basis von den `Joints` für den Kopf und dem Mittelpunkt zwischen den Schultern. Ausgehend vom Punkt in der Mitte dieser beiden `Joints` wird ein Bild aus dem `ColorStream` ausgeschnitten, das genau doppelt so groß wie die Distanz zwischen den `Joints` ist.
Dadurch wird stets ein genau gleich großer Ausschnitt vom Kopf angezeigt, der ungefähr einem Passfoto entspricht. Dabei spielt es keine Rolle, in welcher Position oder Entfernung sich der Spieler vom Sensor befindet. Der Kopf wird die ganze Zeit getrackt und im selben Maßstab angezeigt.

Die Erkennung vom Bildausschnitt erfolgt folgendermaßen:

```
// wenn ein Skeleton verfügbar ist:
Joint halsJoint = skeleton.Joints[JointType.ShoulderCenter];
Joint kopfJoint = skeleton.Joints[JointType.Head];
if (
      (halsJoint.TrackingState == JointTrackingState.Tracked
      || halsJoint.TrackingState ==
JointTrackingState.Inferred)
      && (kopfJoint.TrackingState ==
JointTrackingState.Tracked
      || kopfJoint.TrackingState ==
JointTrackingState.Inferred))
{
      kopfX =
(int)(SkeletonPointToScreen(kopfJoint.Position).X);
      kopfY =
(int)(SkeletonPointToScreen(kopfJoint.Position).Y);
      halsX =
(int)(SkeletonPointToScreen(halsJoint.Position).X);
      halsY =
(int)(SkeletonPointToScreen(halsJoint.Position).Y);
      // Mittelpunkt zwischen Hals und Oberkante Kopf
      kopfXPos = (kopfX + halsX) / 2;
      kopfYPos = (kopfY + halsY) / 2;
      distanzKopfHals = euklDistanz(kopfX, kopfY, halsX,
halsY);
}
// Head Tracked Image zeichnen oder ausblenden
Dispatcher.Invoke(DispatcherPriority.Normal, new Action(() =>
showHeadTrackedImage(skeletonTracked, screenCounter,
colorStream, kopfXPos, kopfYPos, distanzKopfHals * 2)));
```

Das Zeichnen vom Bildausschnitt passiert hier:

```
void showHeadTrackedImage(bool skeletonAvailable, int
imagePos, byte[] imageBytes, int xPos, int yPos, int size){
      if (skeletonAvailable){
            ... (Bild aus ColorStream in Variable wb
      speichern)
            // von Mittelpunkt zu Punkt links oben
            int cropXPos = xPos - size / 2;
```

```
            int cropYPos = yPos - size / 2;
            // Ausschnitt am Bildrand => an Bildkante
        ausrichten
            if (cropXPos < 0) cropXPos = 0;
            if (cropXPos > 640 - size) cropXPos = 640 - size;
            if (cropYPos < 0) cropYPos = 0;
            if (cropYPos > 480 - size) cropYPos = 480 - size;
            // ColorStream croppen und anzeigen
            HeadTrackingImages[imagePos].Source = new
        CroppedBitmap(wb, new Int32Rect(cropXPos, cropYPos,
        size, size));
            HeadTrackingImages[imagePos].Opacity = 1;
        }
        else{ // wenn kein Skelett erkannt, Bild ausblenden
            HeadTrackingImages[imagePos].Opacity = 0;
        }
}
```

4.4 Sprite Animationen

Das verwendete Display scheint sehr anfällig für ein Einbrennen von statischen Inhalten zu sein. Daher war es notwendig, dass sich sämtliche Inhalte kontinuierlich bewegen. Um ein zufällig erscheinendes Zucken unserer Inhalte zu vermeiden entschieden wir uns für eine Animation sämtlicher Inhalte. Solange der Nutzer seine Hände am unteren Rand des Displays hält, pflanzt er Blumen und diese wachsen auch. Die Blumen wiegen im Wind, genauso wie das auch auf einer echten Wiese geschieht. Dieses hin- und herbewegen ist eine Animation, welche zuvor in Adobe After Effects (AE) erstellt wurde. Jede Animation besteht aus 120 Einzelbildern und wiederholt sich nach 4 Sekunden. Die einzelnen Frames wurden als PNG Dateien in AE gerendert. Die Animation der Blumen erfolgt also mit Hilfe einer Sprite Animation.
Bei Programmstart werden die Frames für jede Animation geladen:

```
BitmapSource[,] sprites = new BitmapSource[ NUMBER_OF_SPRITES,
FRAMES_PER_ANIMATION];
//spriteFilenames contains array of folders
//each folder only contains the files for one animation
for (int spriteCount = 0; spriteCount < spriteFilenames.
Length; spriteCount++){
```

```
        string[] files = Directory.GetFiles(spriteFilenames[
        spriteCount]);
        for (int frameCount = 0; frameCount < files.Length;
        frameCount++){
                String file = files[frameCount];
                this.spriteImages[spriteCount, frameCount] = new
                BitmapImage(new Uri(file));
        }}
```

Die Positionierung der Grafiken in WPF erfolgt über die Außenabstände zu allen vier Seiten, also Margins. Um die Bilder nur über X- und Y-Koordinaten zu positionieren, müssen sämtliche Bilder ein HorizontalAlignment=”Left” und VerticalAlignment=”Top” als Eigenschaft gesetzt haben:

```
<Image HorizontalAlignment="Left"
VerticalAlignment="Top"></Image>
```

In unserer Anwendung erzeugen wir neue Blumen im Code, indem wir dem Haupt-Grid dynamisch erzeugte Images als Kindobjekte hinzufügen. Auch hier können die Parameter code-seitig gesetzt werden um so eine einfache Positionierung über X- und Y-Koordinaten zu ermöglichen.

```
private void addSpriteContainer(){
        Image img = new Image();
        img.Width = spriteWidth; //width sprite image
        img.Height = spriteHeight; //height of image
        img.HorizontalAlignment =
        System.Windows.HorizontalAlignment.Left;
        img.VerticalAlignment =
        System.Windows.VerticalAlignment.Top;
        Plant tempPlant = new Plant(img, spriteIndexGlobal++);
        this.FlowerGrid.Children.Add(img);
        this.availableSpriteContainers.Add(tempPlant);
}
```

Die Klasse `Plant` stellt den aktuellen Zustand der Pflanze dar und dadurch auch den der Sprite Animation. So weiß ein Objekt des Typs `Plant`, ob es aktuell gerade am Wachsen oder Verwelken ist, und hält eine Referenz auf den `Image`-Container, welcher dem WPF Layout hinzugefügt wurde. Jede Pflanze ist eindeutig durch eine `PlantId` gekennzeichnet.

Das Rendern der Grafiken erfolgt auf der GPU. Die einzelnen Bilder werden also erst nach dem ersten Anzeigen in den Speicher geladen, bei

Programmstart wird nur eine Referenz geladen. Daher steigt der Speicherbedarf der Anwendung im Programmablauf.
In einem Frame-Thread, welcher alle 30 Millisekunden aufwacht, erfolgt der Austausch der Bilder. Beispielhaft sei dies hier anhand eines `DispatcherTimer` gezeigt. In unserer Anwendung erfolgt dies durch einen separaten Thread, welcher nach jedem Durchlauf für 30ms schläft und dann wieder von vorne beginnt.

```
System.Windows.Threading.DispatcherTimer frameTimer = new
DispatcherTimer();
frameTimer.Tick += OnFrame;
frameTimer.Interval = TimeSpan.FromSeconds(1.0 / 30.0);

private void OnFrame(object sender, EventArgs e){
      this.MyImage.Source = sprites[0, currentFrameCount %
      numberFrames];
      currentFrameCount++;
}
```

Um die Metapher einer Blumenwiese bei unserem System noch zu stärken und um ein Einbrennen der Pflanzenwurzeln zu verhindern, entschieden wir uns dafür, dass die Blumen sich insgesamt nach rechts und links bewegen. Diese Bewegung ändert alle vier Sekunden zufällig die Richtung und die Ausprägung, sodass insbesondere das Einbrennen durch sich wiederholende Muster verhindert wird. Die Positionierung erfolgt durch eine Änderung der Außenabstände:
`img.Margin = new Thickness(img.Margin.Left + xOffsetDiff, img.Margin.Top, 0, 0);`
`xOffsetDiff` ist ein 1-Pixel-Offset welcher in jedem Frame entweder auf 1 oder 0 gesetzt wird, je nachdem wie es für den aktuellen 4-Sekunden-Zyklus notwendig ist. Die Berechnung dessen erfolgt in der Methode `private void calcGlobalOffset()`.

Der Lebenszylus einer Pflanze sieht wie folgt aus:
Sobald ein Nutzer seine Hände am unteren Bildschirmrand positioniert, werden Samen zum Säen in eine Warteschleife eingereiht. Dies erfolgt durch die Methode `private bool queueSeed(int positionX, Color color)`. Falls noch Platz für weitere Pflanzen ist, wird, falls nötig, ein weiterer Image Container erzeugt und dem Haupt-Grid hinzugefügt. Innerhalb des nächsten Zyklus durch den Frametimer werden alle Samen, welche sich in der Warteschleife befinden, “gesät” und somit beginnt die Sprite Animation. Dies

geschieht durch den Aufruf der Methode `private void seedSeeds()`, in welcher der zuvor angelegte `Image` Container einem `Plant` Objekt hinzugefügt wird. Die Bildgröße wird anfangs auf 0 gesetzt. Im weiteren Verlauf wird innerhalb jedes Frames 1. die Bild `Source` ausgetauscht (exemplarische siehe oben) und 2. die Bildgröße geändert, anfangs bis zur maximalen Größe. Wenn kein Nutzer mehr vor dem Display steht, verringert sich die Größe, bis die Pflanze komplett verwelkt ist. Anschließend wird sie aus dem Haupt-Grid entfernt: `private void removePlant(Plant plantToRemove)`
Das Wachstum, Verwelken und Entfernen der Pflanzen erfolgt innerhalb der Methode `private void careForPlants()`.

4.5 Partikelsystem

Anfangs befindet sich im Körperzentrum des Betrachters ein Partikelsystem, welches insbesondere die Aufmerksamkeit im Vorbeilaufen erregen soll. Sobald ein Skelett erkannt wurde, befinden sich an beiden Händen des Schattenbildes Partikelsysteme. Die Partikelsysteme werden durch einem `ParticleSystemManager` verwaltet. Innerhalb dessen wird jedes `ParticleSystem` anhand der Color identifiziert. Sobald ein `ParticleSystem` einmal “gespawnt” (geboren) wurde, entwickeln sich die Partikel an dem Startpunkt über ihre Lebensdauer: das System wächst Anfangs in der Größe, das bedeutet die Partikel breiten sich vom Zentrum aus. Sobald die Partikel ihre maximale Lebensdauer erreicht haben, werden sie kleiner und verschwinden dann schließlich. Sofern ein `ParticleSystem` nicht erneut gespawnt wird, sind keine Partikel mehr erkennbar. Wir spawnen in unserem Programm jedes `ParticleSystem` einmal pro Frame, sofern entsprechende Punkte (z.B. Hand) erkannt wurden. Die alten Partikel leben daher noch so lange bis ihre LiveTime beendet ist. Je nach Voreinstellung ist das innerhalb weniger Sekunden.
Das PartikelSystem wird in der Code-Behind-Datei des MainWindows in der Methode `private void initParticleSystem()` initialisiert.
Anschließend wird innerhalb des Frame-Threads alle 30ms die Methode
`public void SpawnParticle(Point3D position, double speed, Color color, double size, double life)`
für jeden Spawn-Point (zum Beispiel linke Hand, rechte Hand, etc.) aufgerufen. Hierdurch wird die oben genannte Langlebigkeit der Partikel gesichert. Die Dokumentation der Parameter (und empfohlene Werte) finden sich in den Code-Kommentaren.
Zusätzlich muss einmal alle 30ms (d.h. einmal je Frame) die Update Methode des `ParticleSystemManager`s aufgerufen werden, welche als Parameter die

verstrichenen Sekunden seit dem letzten Update Aufruf enthält. Hierdurch wird die nächste Frame-Animation berechnet.

4.6 Anzeige auf dem Public Display in der Amalienstraße 17

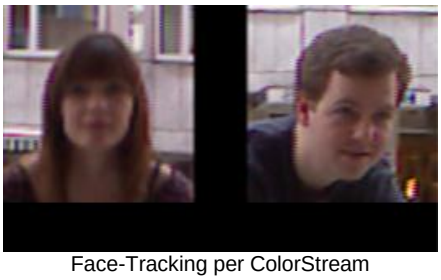
Das Public Display im Schaufenster der Amalienstraße 17 besteht aus vier separaten Displays. Auf diesen wird ein skaliertes Bild in der Auflösung 3412 x 480 angezeigt. Dieses Bild wird aus der Anzeige des primären Displays erzeugt, welcher eine native Auflösung von 1280 x 1024 hat. Um den Skalierungsfaktor zu umgehen arbeiteten wir in unserer Anwendung mit der nativen Auflösung und konnten uns so ein softwareseitiges Skalieren sparen. Das Skalieren auf das Public Display erledigte ohnehin die Software der Displays.
Damit die Anwendung nicht nur auf dem Public Display korrekt dargestellt wird, nehmen die Ausgaben der beiden Kinect-Sensoren jeweils eine relative Breite des Windows ein: Jeder dargestellte Depth-Stream wird auf 50% der Fensterbreite und 100% der Fensterhöhe gestreckt und dann beide in einem zweispaltigen Grid nebeneinander angezeigt.

5. Zusammenfassung

StarFlowers ist eine interaktive Kunstsimulation, welche eine kurze Interaktion auf Public Displays ermöglicht. Passanten werden durch ihren Schatten, welcher von einem Partikelsystem umgeben ist, animiert sich zum Display zu drehen. Sobald sie erfasst wurden, zeigt das System an, dass sie mit Hilfe ihrer Hände zwei Partikelsysteme zum unteren Bildschirmrand bewegen müssen. Dort können sie Blumen pflanzen, welche so lange wachsen, wie der Benutzer sich nach unten beugt. Wenn das Display wieder verlassen wird, gehen die Pflanzen langsam ein, bis der Ausgangszustand wieder hergestellt ist.

Insgesamt läuft unsere Anwendung stabil auf den Displays. Durch verschiedene Techniken wird ein Einbrennen von Inhalten verhindert. Somit ist die Applikation auch auf einen Dauerbetrieb ausgelegt.
Das Tracking von zwei Personen und die Zusammenarbeit von zwei Kinect Sensoren ist stabil. Die grafische Darstellung der Inhalte funktioniert ohne Probleme mit 33 Frames pro Sekunde, experimentell sind über 60 FPS möglich. Dies ist möglich, da die Inhalte größtenteils auf der GPU gerendert werden und die Anzeige, losgelöst von den rechenintensiven Skelett- und Tiefenbildberechnungen, in einem separaten Thread erfolgt.

6. Bilder



Face-Tracking per ColorStream

mehrere User bedienen die Simulation

Blumenlandschaft auf dem Public Display

Partikelsysteme in den Händen

Video, welches die Anwendung demonstriert: http://vimeo.com/64384008