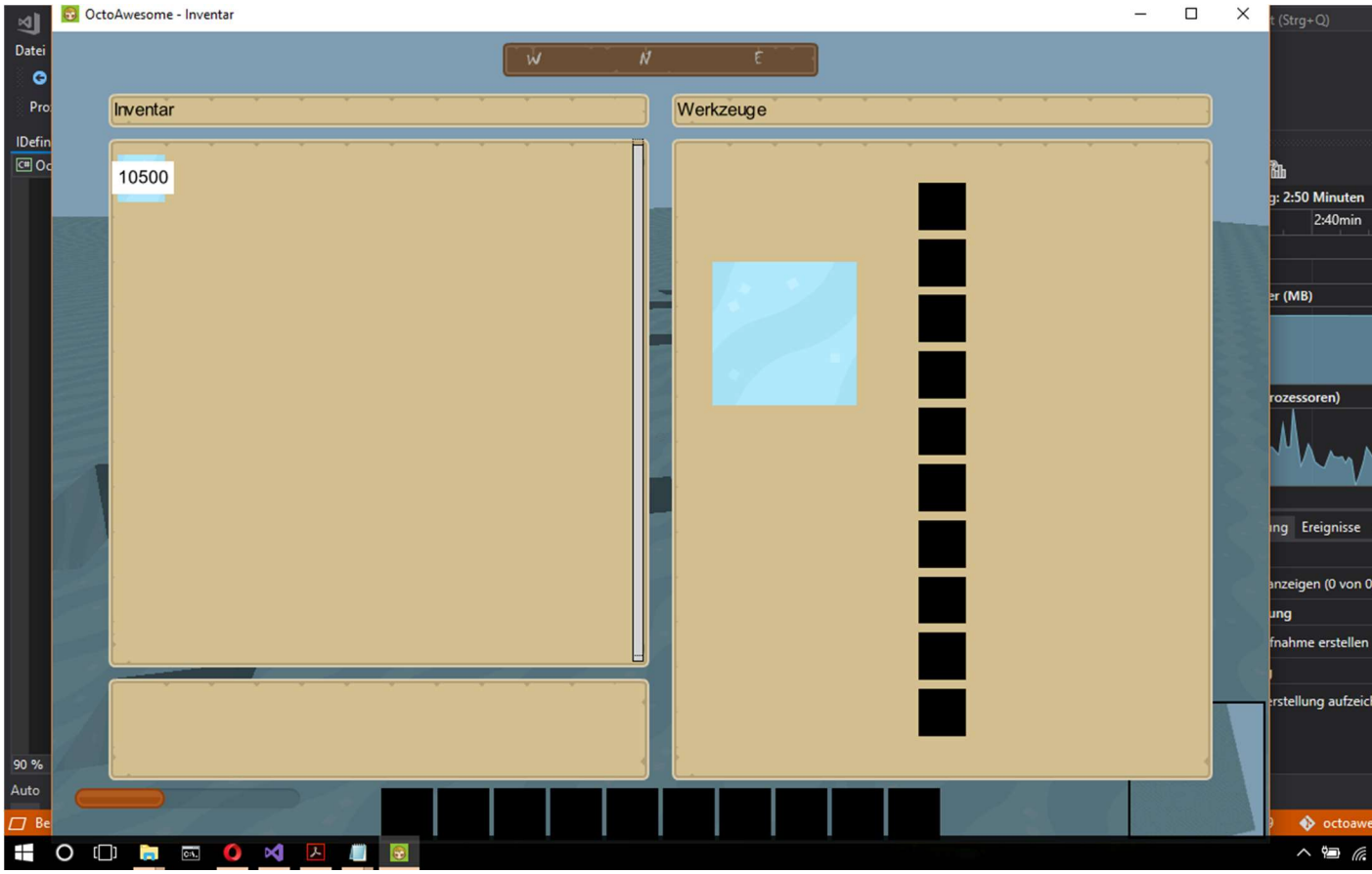
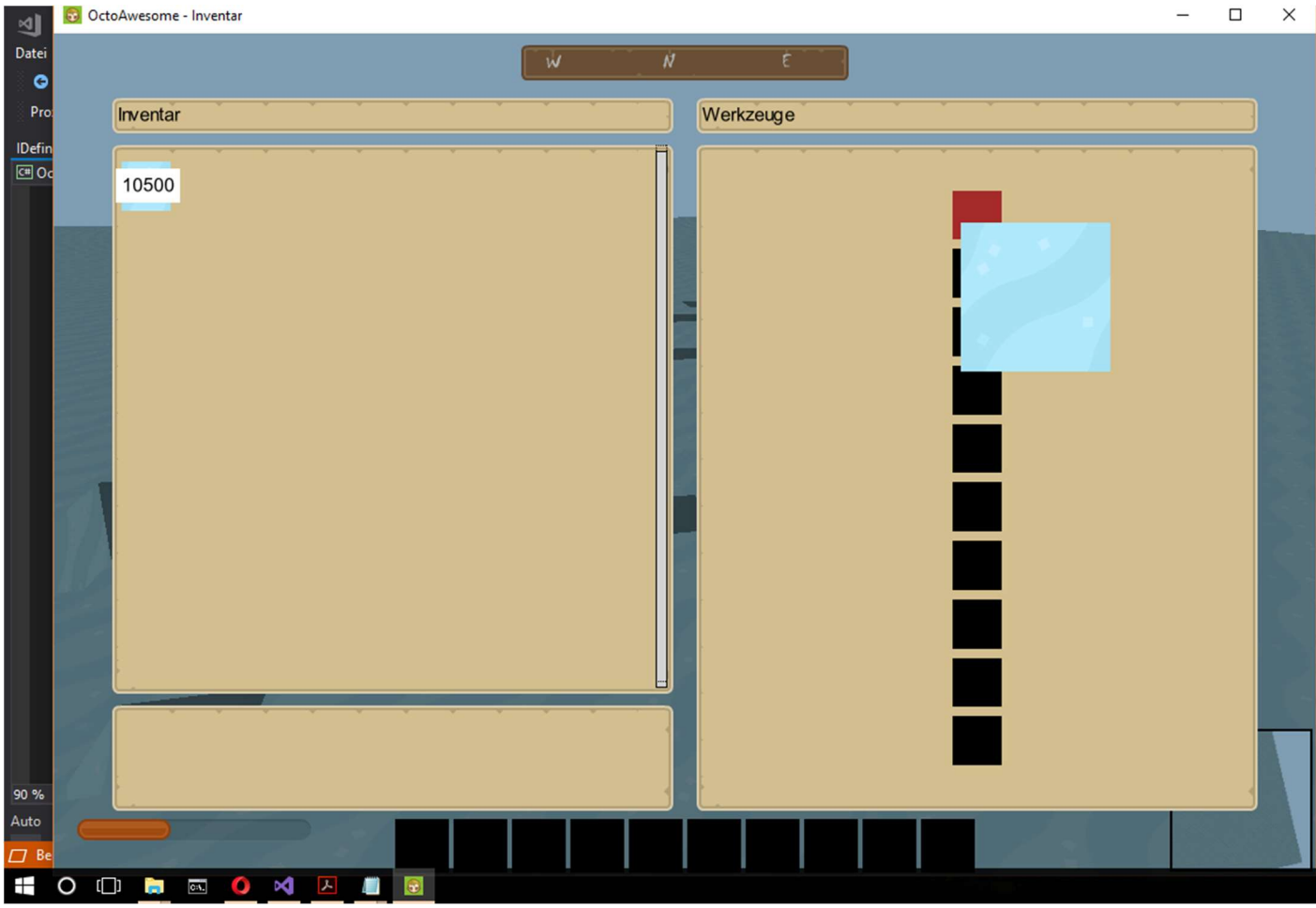

OctoAwesome - Inventar
W N E
Inventar
Werkzeuge
10500


OctoAwesome - Inventar
W N E
Inventar
Werkzeuge
10500

OctoAwesome.dll

Simulation Klasse wurde an Änderungen angepasst und aufgeräumt. Entity Komponenten wurden entfernt oder verschoben. Die Component Klasse hat einen neuen Konstruktor erhalten und wurde erweitert, zusätzlich wurde das ISerializable Interface implementiert, ebenso wie bei der Entity Klasse. Außerdem wurde die Entity Klasse überarbeitet. An der Logik dieser Klasse hat sich nichts verändert.

```
/// <summary>
/// Entity die regelmäßig eine Updateevent bekommt
/// </summary>
/// <param name="needUpdate">Indicates that Entity need an Update.</param>
public Entity(bool needUpdate)…
/// <summary>
/// Indicates that the <see cref="Entity"/> need an Update.
/// </summary>
public bool NeedUpdate { get; }
/// <summary>
/// Horizontaler winkel.
/// </summary>
public float Azimuth { get; private set; }
/// <summary>
/// <see cref="Coordinate"/> of the <see cref="Entity"/>.
/// </summary>
public Coordinate Position { get; private set; }
/// <summary>
/// Updatemethod for the <see cref="Entity"/>
/// </summary>
/// <param name="gameTime">Time of the Simulation.</param>
/// <param name="service">Game services</param>
internal void Update(GameTime gameTime, IGameService service)
{
    OnUpdate(gameTime, service);
}
/// <summary>
/// Called dirung Update of the <see cref="Entity"/>
/// </summary>
/// <param name="gameTime">Time of the Simulation.</param>
/// <param name="service">Game services</param>
protected virtual void OnUpdate(GameTime gameTime, IGameService service)…
}
/// <summary>
/// Set the Postion of the <see cref="Entity"/>
/// </summary>
/// <param name="position">The new Position</param>
/// <param name="azimuth">Horizontal angle</param>
public void SetPosition(Coordinate position, float azimuth = 0)
{
    if(Cache != null)
        Cache.SetCenter(Cache.Planet, new Index2(Position.ChunkIndex));

    OnSetPosition(position, azimuth);
    Position = position;
    Azimuth = azimuth;
}
/// <summary>
/// Called during SetPosition (before Position is set)
/// </summary>
/// <param name="position">New Position</param>
/// <param name="azimuth">Horizontal angle</param>
protected virtual void OnSetPosition(Coordinate position, float azimuth)…
```

Das IResourceManager Interface in der Entity Initialize Methode wurde durch das IGameService Interface ersetzt. Zur EntityComponent Klasse kamen ein neuer Konstruktor sowie ein NeedUpdate Property dazu. Das Konzept von Entity Komponenten wurde geändert sodass sie nicht nur eine Datensammlung darstellt. Sondern sich auch selbst berechnen.

```
/// <summary>
/// <see cref="Entities.Entity"/> of this <see cref="EntityComponent"/>
/// </summary>
public Entity Entity { get; private set; }
/// <summary>
/// Constructor of <see cref="EntityComponent"/>
/// </summary>
/// <param name="needupdate">Indicates that the EntityComponent need updates</param>
public EntityComponent(bool needupdate) : base(needupdate)…
/// <summary>
/// Default Constructor
/// </summary>
public EntityComponent() : base()…
/// <summary>
/// Set the Entity for this Component
/// </summary>
/// <param name="entity"></param>
public void SetEntity(Entity entity)
{
    if (entity == null) return;
    if (Entity != null) return;
    OnSetEntity(entity);
    Entity = entity;
}
/// <summary>
/// Called during SetEntity
/// </summary>
/// <param name="entity"></param>
protected virtual void OnSetEntity(Entity entity)…
```

Zu diesem Zeitpunkt kann ich das Komponentensystem so wie ich es mir vorgestellt habe nicht weiter entwickeln. Dazu sind Änderung am Persistancesystem und Definitionsystem notwendig. Die beiden Punkte stehen nach dem hoffentlich (diesmal) erfolgreichen Push als nächstes auf der Liste. Das Grundlegende Interface existiert bereits. Allerdings ist dies nicht die finale Fassung, sondern ein zwischen schritt. Zu viel auf einmal tut selten gut.

```
/// <summary>
/// Interface for Serializable classes.
/// </summary>
public interface ISerializable
{
    /// <summary>
    /// Serialisiert die Entität mit dem angegebenen BinaryWriter.
    /// </summary>
    /// <param name="writer">Der BinaryWriter, mit dem geschrieben wird.</param>
    /// <param name="definition">Der aktuell verwendete IDefinitionManager </param>
    void Serialize(BinaryWriter writer, IDefinitionManager definition);
    /// <summary>
    /// Deserialisiert die Entität aus dem angegebenen BinaryReader.
    /// </summary>
    /// <param name="reader">Der BinaryWriter, mit dem gelesen wird.</param>
    /// <param name="definition">Der aktuell verwendete IDefinitionManager </param>
    void Deserialize(BinaryReader reader, IDefinitionManager definition);
}
```

Die ComponentList hat für schnellere Zugriffe, Überladungen für Methoden erhalten und wurde erweitert. Ebenfalls wurde eine Update Methode implementiert die alle Komponenten Updated. Beim Hinzufügen von Komponenten in die Liste werden alle Komponenten die ein Update brauchen zusätzlich in eine extra Liste geschrieben.

```
/// <summary>
/// Default Constructor of <see cref="ComponentList{T}"/>.
/// </summary>
/// <param name="onAdd">On <see cref="{T}"/> add delegate.</param>
/// <param name="onRemove">On <see cref="{T}"/> remove delegate.</param>
public ComponentList(Action<T> onAdd, Action<T> onRemove) : this()…
/// <summary>
/// Adds a new Component to the List and replace Items with the same Key.
/// </summary>
/// <typeparam name="V">Type of Component.</typeparam>
/// <param name="type">Type.</param>
/// <param name="component">Component.</param>
/// <param name="replace">Replace i already inclueded.</param>
public void AddComponent<V>(Type type, V component, bool replace) where V : T…
/// <summary>
/// Update all updatealbe components in this <see cref="ComponentList{T}"/>
/// </summary>
/// <param name="gameTime">Simulation time.</param>
/// <param name="service">Game Service</param>
public void Update(GameTime gameTime, IGameService service)…
/// <summary>
/// Checks if the <see cref="Component"/> is included.
/// </summary>
/// <param name="type"><see cref="Type"/> of the Component param>
/// <returns></returns>
public bool ContainsComponent(Type type)…
/// <summary>
/// Returns the <see cref="Component"/> of the given Type or null
/// </summary>
/// <typeparam name="V">Component Type</typeparam>
/// <param name="type">Type.</param>
/// <returns></returns>
public V GetComponent<V>(Type type) where V : class…
/// <summary>
/// Try to get a <see cref="Component"/>.
/// </summary>
/// <typeparam name="V"><see cref="Type"/> of Component typeparam>
/// <param name="type">Type</param>
/// <param name="component">Component</param>
/// <returns></returns>
public bool TryGetComponent<V>(Type type, out V component) where V : class…
/// <summary>
/// Removes the <see cref="Component"/> of the given Type.
/// </summary>
/// <param name="type"><see cref="Type"/> of the Component </param>
/// <returns></returns>
public bool RemoveComponent(Type type)…
```

Um eine Entity steuern zu können benötigt sie Inputs und diese bekommt sie von einem Kontroller aber damit das System weiß ob die Entity kontrolliert werden kann wurde das IControllable Interface eingeführt. Auch dies ist nur ein zwischen Schritt aber ein allgemeiner Ansatz. Es bietet Informationen über die Lage der Entity auf der Welt sowie eine Methode um diese festzulegen als auch ein Controller Property und Methoden um den Kontroller zu setzten bzw. zurückzusetzen. Die Berechnung der Bewegung kann im Controller erfolgen oder in einer dafür gedachten Komponente.

```
/// <summary>
/// Interface for controllable entities (Dogs, Cats, etc.)
/// </summary>
public interface IControllable
{
    /// <summary>
    /// Horizontal angle of the Entity.
    /// </summary>
    float Azimuth { get; }
    /// <summary>
    /// Position in the world
    /// </summary>
    Coordinate Position { get; }
    /// <summary>
    /// Controller
    /// </summary>
    IEntityController Controller { get; }
    /// <summary>
    /// Register a controller
    /// </summary>
    /// <param name="controller">Controller to register</param>
    void Register(IEntityController controller);
    /// <summary>
    /// Reset the controller
    /// </summary>
    void Reset();
    /// <summary>
    /// Set the Position of the Entity.
    /// </summary>
    /// <param name="position">New Position.</param>
    /// <param name="azimuth">Horizontal angle of the <see cref="Entity"/></param>
    void SetPosition(Coordinate position, float azimuth);
}
```

Den Gegenpart zum IControllable Interface stellt das IEntityController da. Es biete eine wenn auch zur Zeit nicht vollständige Möglichkeit zwischen der UserEntity und dem realen Spieler Informationen auszutauschen. Einfachen Tasten (Slot, Jump, Interact und Apply - Buttons) die aktive im Code quittiert werden müssen, sowie der selektierte Block, Blockseite etc. und Inputs die eine Maussteuerung ermöglichen. Das Interface ist von der Ursprünglichen Implementierung inspiriert. (Nicht das einer sagt ich klaue Code).

```
/// <summary>
/// Pressed numbers on Keyboard (0 to 9)
/// </summary>
bool[] SlotInput { get; }
/// <summary>
/// Indicates that Slots have to be shifted left
/// </summary>
bool SlotLeftInput { get; set; }
/// <summary>
/// Indicates that Slots have to be shifted right
/// </summary>
bool SlotRightInput { get; set; }
/// <summary>
/// Input for interaction
/// </summary>
bool InteractInput { get; set; }
/// <summary>
/// Input for apply
/// </summary>
bool ApplyInput { get; set; }
```

```
/// <summary>
/// Input for Jump
/// </summary>
bool JumpInput { get; set; }
/// <summary>
/// Tilt feedback in world coordinatesystem
/// </summary>
float Tilt { get; set; }
/// <summary>
/// Yaw feedback in wolrd coordinatesystem
/// </summary>
float Yaw { get; set; }
/// <summary>
/// Roll feedback in wolrd coordinatesystem
/// </summary>
float Roll { get; set; }
/// <summary>
/// Move direction in Enttiy coordinatesystem
/// </summary>
Vector2 MoveInput { get; }
/// <summary>
/// Head input in Entity coordinatesystem
/// </summary>
Vector2 HeadInput { get; }
/// <summary>
/// Index of Block to interact
/// </summary>
Index3? SelectedBlock { get; }
/// <summary>
/// Selected point of the block
/// </summary>
Vector2? SelectedPoint { get; }
/// <summary>
/// Selected side of the block
/// </summary>
OrientationFlags SelectedSide { get; }
/// <summary>
/// Selected edge o the block
/// </summary>
OrientationFlags SelectedEdge { get; }
/// <summary>
/// Selected corner of the block
/// </summary>
OrientationFlags SelectedCorner { get; }
/// <summary>
/// Indicats that the controller can be freezed
/// </summary>
bool CanFreeze { get; }
/// <summary>
/// Indicates that the controller is freezed or freez the controller
/// </summary>
bool Freezed { get; set; }
```

Das Interface muss noch überarbeitet werden sodass eine 2 und 3D Steuerung möglich ist. Anschließend eine Basisinterface einführen oder dasselbe Interface KI tauglich machen. Ich würde Variante zwei bevorzugen. Zusätzlich werden Entitys zukünftig von IEntityDefinition´s definiert. Ermöglicht eine generischere Population der Welt sowie besseres Persistierungsverhalten der Entitäten. Ich verspreche mir davon außerdem eine bessere Handhandhabung von Entitäten sowohl bei der Entwicklung, Wartung als auch Erweiterung des Spiels. Wie die Definition aussehen könnte, kann ich leider noch nicht sagen aber die Definition könnte sich am System.Type orientieren. Des Weiteren werden die Definitionen zukünftig nicht mehr oder nicht nur als eigene Klassen exerzieren

sondern auch als Files auf der Festplatte oder in der Clound. Eine Serialisierung von Implementierten Definitons wird auch implementiert. Die ist von meiner Seite mit jason und xml geplant. Zurzeit sieht die IEntityDefinition folgendermaßen aus, es existiert aber noch keine Implementierung die Verwendet wird.

```
/// <summary>
/// <see cref="IEntityDefinition"/> defines an <see cref="Entity"/>
/// </summary>
public interface IEntityDefinition : IDefinition
{
    /// <summary>
    /// Runtime type of the Entity.
    /// </summary>
    Type AssosiatedType { get; }
    /// <summary>
    /// Indicats that this definition can be stored in an inventory.
    /// </summary>
    bool IsInventoryable { get; }
    /// <summary>
    /// Modelname of <see cref="Entity"/>
    /// </summary>
    float RotationZ { get; }
    /// <summary>
    /// Mass of <see cref="Entity"/>
    /// </summary>
    float Mass { get; }
    /// <summary>
    /// Radius of the Entity
    /// </summary>
    float Radius { get; }
    /// <summary>
    /// Height of the Entity
    /// </summary>
    float Height { get; }
    /// <summary>
    /// Returns a object of type T.
    /// </summary>
    /// <typeparam name="T">Type of object</typeparam>
    /// <param name="name">internal key</param>
    /// <param name="resource">resource</param>
    /// <returns></returns>
    bool TryGetResource<T>(string name, out T resource);
}
```

Um Entitäten rendern zu können wurde ein IDrawable Interface eingeführt. Dieses ist momentan nur Temporär und übergangsweise Definiert und verhält sich genauso wie die früher RenderComponent. Das hat den Grund, dass eine starke Abhängigkeit zur IEntityDefinition besteht die nur als Platzhalter existiert. Ich würde gern unsere Graphikexperten bitten die Definition vorzunehmen. Den Code spare ich mir an der Stell. Ein wesentlich interessanteres Thema dürfte das Interface IGameService sein. Ich hab es mit der Absicht erstellt um wiederverwendbaren Code auszulagern, Methoden für Moder bereitzustellen, die aufwendig sind und nicht unbedingt selbst implementiert werden müssen. Hierfür werden noch weitere Interfaces kommen, dass erste Interface das simultan zu IGameService eingeführt wurde ist IInventory. Es ermöglicht eine allgemeine Handhabung von Inventar Implementierungen (zumindest was das ein- und auslagern betrifft). Im Entwicklungsverlauf sollen noch Dienste hinzukommen die nicht selbst implementiert werden könne. Darunter könnte Serverkommunikation fallen, für Aktionshäuser oder Ingame Händler die ihren Warenbestand Synchronisieren müssen. Es bleibt spannend. Zusätzlich implementiert das IGameService Interface das

IServiceProvider Interface sodass es durch Extensions erweitert werden kann. Dieses Feature ist allerdings noch nicht implementiert.

```
/// <summary>
/// DefinitionManager
/// </summary>
IDefinitionManager DefinitionManager { get; }
/// <summary>
/// Take a Block from the World.
/// </summary>
/// <param name="controller"> IEntityController of the Entity </param>
/// <param name="cache"> ILocalChunkCache of the Entity </param>
/// <param name="block"><see cref="IInventory"/> of the <see cref="Entity"/></param>
/// <returns>true if the block was successully taken</returns>
bool TakeBlock(IEntityController controller, ILocalChunkCache cache,
     out IInventoryableDefinition block);
/// <summary>
/// Take a Block from the World and add it to the invetory.
/// </summary>
/// <param name="controller"> IEntityController of the Entity </param>
/// <param name="cache"> ILocalChunkCache of the <see cref="Entity"/></param>
/// <param name="inventory">Inventory to store the item</param>
/// <returns>true if the block was successully taken</returns>
bool TakeBlock(IEntityController controller, ILocalChunkCache cache,
     IInventory inventory);
/// <summary>
/// Instanziert einen neuen local Chunk Cache.
/// </summary>
/// <param name="dimensions">Größe des Caches in Zweierpotenzen</param>
/// <param name="range">Gibt die Range in alle Richtungen an.</param>
/// <param name="passive">Indicats that the Cache is passive param>
ILocalChunkCache GetLocalCache(bool passive, int dimensions, int range);
/// <summary>
/// Set a block or interact with a Tool.
/// </summary>
/// <param name="position"> Coordinate of the Entity </param>
/// <param name="controller"> IEntityController of the Entity</param>
/// <param name="cache">ILocalChunkCache of the <see cref="Entity"/></param>
/// <param name="slot">Used <see cref="InventorySlot"/></param>
/// <param name="inventory"> IInventory of the Entity </param>
/// <param name="height">Height of the <see cref="Entity"/></param>
/// <param name="radius">Radius of the <see cref="Entity"/></param>
/// <returns>true if the interaction was successful</returns>
bool InteractBlock(Coordinate position, float radius, float height,
     IEntityController controller, ILocalChunkCache cache, InventorySlot slot,
     IInventory inventory);
/// <summary>
/// Calculates the collison between an <see cref="Entity"/> and the world.
/// </summary>
/// <param name="gameTime">Simulation time</param>
/// <param name="position">Position of the <see cref="Entity"/></param>
/// <param name="cache"><see cref="ILocalChunkCache"/> as workspace</param>
/// <param name="radius">radius of the <see cref="Entity"/></param>
/// <param name="height">height of the <see cref="Entity"/></param>
/// <param name="deltaPosition">Calculated delta position for this cycle</param>
/// <param name="velocity">claculated velocity for this cycle</param>
/// <returns>the velocity of the Entity after collision checks</returns>
Vector3 WorldCollision(GameTime gameTime, Coordinate position, ILocalChunkCache
        cache, float radius, float height, Vector3 deltaPosition, Vector3 velocity);
```

```
/// <summary>
/// Fügt ein Element des angegebenen Definitionstyps hinzu.
/// </summary>
/// <param name="definition">Die Definition.</param>
void AddUnit(IInventoryableDefinition definition);
/// <summary>
/// Entfernt eine Einheit vom angegebenen Slot.
/// </summary>
/// <param name="slot">Der Slot, aus dem entfernt werden soll.</param>
/// <returns>Gibt an, ob das entfernen der Einheit aus dem Inventar funktioniert
hat. False, z.B. wenn nicht genügend Material bzw. Blöcke vorhanden sind-</returns>
bool RemoveUnit(InventorySlot slot);
```

Hinzu kommt noch ein IUserinterfaceExtension Interface, das ermöglicht Extensions bzw. Komponenten (Erweiterungen) für den Player Charakter im Userinterface zu registrieren und Anzuzeigen. Es ist dazu gedacht es in abgeleitet EntityComponet Klassen zu implementieren. Es definiert nur eine Methode die eine IUserInterfaceExtensionManager entgegen nimmt. Durch den ein oder mehrer UI – Element registriert werden können. Das IUserInterfaceExtensionManager Interface definiert die für die Registrierung notwendigen Methoden. Momentan ist es noch nicht möglich Element während des Spiels zu registrieren, dass Erfolg sobald der Spieler Charakter geladen wurde. Das Interface muss noch soweit erweitert werden, dass Screens gewechselt werden können. Mögliche Ziele für die Registrierung sind der GameScreen, InventoryScreen und das DebugControl.

```
/// <summary>
/// Definitionmanager
/// </summary>
IDefinitionManager DefinitionManager { get; }
/// <summary>
/// Register an <see cref="IUserInterfaceExtension"/> to Gamescreen
/// </summary>
/// <param name="controltype">Type of the Control</param>
/// <param name="args">contructor parameter, [reserved parameter] +
                       args</param>
/// <returns></returns>
bool RegisterOnGameScreen(Type controltype, params object[] args);
/// <summary>
/// Register an <see cref="IUserInterfaceExtension"/> to InvertoryScreen
/// </summary>
/// <param name="controltype">Type of the Control</param>
/// <param name="args">contructor parameter, [reserved parameter] +
                       args</param>
/// <returns></returns>
bool RegisterOnInventoryScreen(Type controltype, params object[] args);
/// <summary>
/// Register an Debug label to the Game debug Control
/// </summary>
/// <param name="right"></param>
/// <param name="updatefunc"></param>
/// <returns></returns>
bool RegisterOnDebugScreen(bool right, Func<string> updatefunc);
/// <summary>
/// Load Textures for UI
/// </summary>
/// <param name="type">Texture type</param>
/// <param name="key">Texture key</param>
/// <returns></returns>
Texture2D LoadTextures(Type type, string key);
```

```
/// <summary>
/// Interface for UI dependend <see cref="EntityComponent"/>
/// </summary>
public interface IUserInterfaceExtension
{
    /// <summary>
    /// Register the extending UI.
    /// </summary>
    /// <param name="manager">UI-Manager</param>
    void Register(IUserInterfaceExtensionManager manager);
}
```

Außerdem wurde das IDefinitionManager Interface erweitert. Hinzu kamen nur einige Methoden. Im Zuge der Überarbeitung wie oben Erwähnt wird näher darauf eingegangen. Sowie ein weiterer Platzhalter das IResourceDefinition Interface. Eine kleine aufräum Aktion und Überladungen für Methoden die Sinnvoll sind auch wenn sie der Namensgebung wiedersprechen! Tja wenn der Name falsch ist kann ich auch nichts dafür.

OctoAwesomeBasics.dll

Im Zuge der Reformation wurden alle alten außer zwei Entity Komponenten entfernt. Die ToolBarComponent und InventorComponent durften bleiben. Bei beiden wurde das IUserInterfaceExtension Interface implementiert um die alte Funktionalität aufrecht zu erhalten. Das Selektieren von Blöcken im Inventar wurde jedoch nicht wieder umgesetzt. Der ComplexPlantedGenerator Code wurde angepasst. Keine Sorge nicht der wichtige Teil. CollisionPlanes wurden in die OctoAwesome.dll verschoben für allgemeine Zugänglichkeit. Als Ersatz für die alten Simulatins Komponenten wurde die GroundPhysicComponent hinzugefügt. Die alten SimulationComponent Klassen wurden auch entfern und durch eine neue erstetzt. Alle möglichen overrides sind unten sichtbar.

```
public float Force { get; }
public float Mass { get; }
public float Radius { get; }
public float Height { get; }

public bool OnGround { get; private set; }
public float Azimuth { get; private set; }
public Vector3 Inputdirection { get; private set; }
public Vector3 Inputforce { get; private set; }
public Vector3 Forces { get; private set; }
public Vector3 Acceleration { get; private set; }
public Vector3 Velocity { get; private set; }
public Vector3 DeltaPosition { get; private set; }
public GroundPhysicComponent(float mass, float force, float radius, float height) :
   base(true)
{
    Mass = mass;
    Force = force;
    Radius = radius;
    Height = height;
}
public void Register(IUserInterfaceExtensionManager manager)
{
    manager.RegisterOnGameScreen(typeof(HealthBarControl), "");
}
```

```
protected override void OnSetEntity(Entity entity)
{
    IPlanet planet = entity.Cache.GetPlanet(entity.Position.Planet);
    entity.Position.NormalizeChunkIndexXY(planet.Size);
    entity.Cache.SetCenter(planet, new Index2(entity.Position.ChunkIndex));
}
public override void Update(GameTime gameTime, IGameService service)
{
    IEntityController controller = (Entity as IControllable)?.Controller;
    if (controller != null)
        CalcControllerInpu(gameTime, controller);
    else Inputforce = Vector3.Zero;
    //TODO: entity colliosion
    CalcMotion(gameTime);
    Velocity = service.WorldCollision(gameTime, Entity.Position, Entity.Cache,
    Radius, Height, DeltaPosition, Velocity);
    // TODO: Was ist für den Fall Gravitation = 0 oder im Scheitelpunkt des Sprungs?
    OnGround = Velocity.Z == 0;
    if (Velocity.IsEmtpy()) DeltaPosition = Vector3.Zero;
    else
    {
        DeltaPosition = Velocity * (float) gameTime.ElapsedGameTime.TotalSeconds;
        Azimuth = MathHelper.WrapAngle((float) Math.Atan2(Velocity.Y, Velocity.X));
        Entity.SetPosition(Entity.Position + DeltaPosition, Azimuth);
    }
}
private void CalcControllerInpu(GameTime gameTime, IEntityController controller)
{
    controller.Yaw += (float) gameTime.ElapsedGameTime.TotalSeconds *
    controller.HeadInput.X;
    controller.Tilt = Math.Min(1.5f, Math.Max(-1.5f, controller.Tilt + (float)
    gameTime.ElapsedGameTime.TotalSeconds * controller.HeadInput.Y));

    // calculate dircetion of motion
    float lookX = (float) Math.Cos(controller.Yaw);
    float lookY = -(float) Math.Sin(controller.Yaw);
    Inputdirection = new Vector3(lookX, lookY, 0) * controller.MoveInput.Y;

    float stafeX = (float) Math.Cos(controller.Yaw + MathHelper.PiOver2);
    float stafeY = -(float) Math.Sin(controller.Yaw + MathHelper.PiOver2);
    Inputdirection += new Vector3(stafeX, stafeY, 0) * controller.MoveInput.X;

    Inputforce = Inputdirection * Force;

    // DOTO: on ground einbauen.
    if (controller.JumpInput)
    {
        controller.JumpInput = false;
        Velocity += new Vector3(0, 0, 5);
    }
}
```

Die Logik wurde im Wesentlichen beibehalten lediglich die Berechnungen unterscheiden sich. Diese sind allerdings nicht besonders gut ausgeführt. Die Berechnungen weichen stark von der alten Implementierung ab, sodass diese wahrscheinlich wieder eingeführt wird. Es wurde eine andere Physikalische Herangehensweise verwendet. Die Beschleunigung erfolgt ausschließlich über Kräfte die maximal Geschwindigkeit wird durch Reibung und Kraft der Entity bestimmt. Die Handhabbarkeit hat sehr stark gelitten. Es muss ja nicht realistisch sein sondern Funktionieren.

```
private void CalcMotion(GameTime gameTime)
{
    Forces = Vector3.Zero;
    float gravity = Entity.Cache.Planet.Gravity;
    if (gravity == 0) gravity = 9.81f;
    float gravityforce = Mass * gravity;

    Index3 blockPos = new Index3(0, 0, -1) + Entity.Position.GlobalBlockIndex;
    ushort groundblock = Entity.Cache.GetBlock(blockPos);
    float fluidC = 0;
    float stictionC = 0;
    float slideC = 0;
    float airCoefficent = 5;
    if (groundblock != 0 && OnGround)
    {
        //TODO: friction of underground
        //IBlockDefinition definition =
        //definitionManager.GetDefinitionByIndex(groundblock) as IBlockDefinition;
        //if (definition != null)
        //{
        //}
        slideC = 0.1f;
        stictionC = 0.3f;
        fluidC = 25;
        if (Inputforce.IsEmtpy()) fluidC *= 10f;
    }
    float xfrictionforce = 0;
    float yfrictionforce = 0;
    if(Velocity.LengthSquared <= 0.01f)
    {
        Velocity = Vector3.Zero;
        //stiction
        float stiction = stictionC * gravityforce;
        if (stiction < Math.Abs(Inputforce.X))
            xfrictionforce = Math.Sign(Inputforce.X) * stiction;
        if (stiction < Math.Abs(Inputforce.Y))
            yfrictionforce = Math.Sign(Inputforce.Y) * stiction;
        Forces = new Vector3(xfrictionforce, yfrictionforce, gravityforce);
    }
    else
    {
        //sliding friction
        float slidefriction = slideC * gravityforce;
        if (slidefriction < Math.Abs(Inputforce.X))
            xfrictionforce = Math.Sign(Velocity.X) * slidefriction;
        if (slidefriction < Math.Abs(Inputforce.Y))
            yfrictionforce = Math.Sign(Velocity.Y) * slidefriction;
        //fluid friction
        Forces += new Vector3(Velocity.X * fluidC + xfrictionforce, Velocity.Y *
             fluidC + yfrictionforce, gravityforce);
    }
    // air friction
    // F_a = 1/2 c_w A roh v^2
    float velo = Velocity.Length;
    Vector3 normal = Velocity.Normalized();
    Forces += 0.5f * airCoefficent * Velocity;

    Acceleration = (Inputforce - Forces) / Mass;
    Velocity += Acceleration * (float) gameTime.ElapsedGameTime.TotalSeconds;
    DeltaPosition = Velocity * (float) gameTime.ElapsedGameTime.TotalSeconds;
}
```

Die nötigen Control´s für die Userinterface Erweiterung wurden aus OctoAwesome.Client.exe verschoben und angepasst. Vollständige Initialisierung im Konstruktor etc. Alle drei Komponenten erweitern das UI. ToolBarControl, InventoryControl und HealthbarControl bei der GroundPhysicComponent. Außerdem wurde ein weiteres Control für den InventarScreen der ToolBarComponent implementiert. Bei der WauziEntity wurden die IControllable und IDrawalbe Interfaces implementiert. Die TreeDefinition´s wurden an das neue IDefinitionManager Interface angepasst. Die Extension Klasse wurde in zwei Teile aufgeteilt: WorldExtension dazu gehören die Generatoren und BlockDefinition´s und PhysicExtension die die Entity Komponenten besteuert.

OctoAwesome.Client.exe

In der PlayerComponent wurde das IEntityController Interface implementiert bei der Implementierung wurden diese und die GameScreen angepasst. Die GameScreen OnUpdate Methode:

```
Vector2 move = Vector2.Zero;
if (pressedMoveUp) move += new Vector2(0f, 1f);
if (pressedMoveLeft) move += new Vector2(-1f, 0f);
if (pressedMoveDown) move += new Vector2(0f, -1f);
if (pressedMoveRight) move += new Vector2(1f, 0f);
Manager.Player.MoveInput = move;

Vector2 head = Vector2.Zero;
if (pressedHeadUp) head += new Vector2(0f, 1f);
if (pressedHeadDown) head += new Vector2(0f, -1f);
if (pressedHeadLeft) head += new Vector2(-1f, 0f);
if (pressedHeadRight) head += new Vector2(1f, 0f);
Manager.Player.HeadInput += head;

HandleGamePad();

base.OnUpdate(gameTime);
```

Bei der PlayerComponent wurde die OnUpdate Metho vollständig entfernt. Die Änderungen in der SetEntity Methode Sehen wie folg aus.

```
CurrentEntity = entity;
if (CurrentEntity != null && CurrentEntity is IControllable current)
    current.Reset();
if (entity is IControllable controllable)
    controllable.Register(this);
if (CurrentEntity is Entities.IDrawable draw)
    HeadOffset = new Vector3(0, 0, draw.Height - 0.3f);
else HeadOffset = new Vector3(0, 0, 3.2f);

if (CurrentEntity != null)
{
    foreach(Entities.EntityComponent comp in entity.Components)
        if (comp is IUserInterfaceExtension extension)
            extension.Register(Game.Screen);
}
else
{
    Game.Screen.CleanExtensions();
}
```

Die Logik der Update Methode wurde nach GroundPhysicComponent und ToolbarComponent verschoben.

In der ScreenComponent wurde das IUserInterfaceExtensionManager Interface implementiert. Die Klasse wurde entsprechend angepasst.

```
public IEnumerable<Func<Control>> GameScreenExtension
    { get { return gamescreenextension; } }
public IEnumerable<Func<Control>> InventoryScreenExtension
    { get { return inventoryscreenextension; } }
public IEnumerable<(bool right, Func<string> updatefunc)> DebugControlExtensions
    { get { return debugscreenextensions; } }

private List<Func<Control>> gamescreenextension;
private List<Func<Control>> inventoryscreenextension;
private List<(bool, Func<string>)> debugscreenextensions;

public bool RegisterOnGameScreen(Type controltype, params object[] args)
{
    if (Check(controltype, args, out object[] parameter))
    {
        gamescreenextension.Add(() => InternalCreate(controltype, parameter));
        return true;
    }
    return false;
}
public bool RegisterOnInventoryScreen(Type controltype, params object[] args)
{
    if (Check(controltype, args, out object[] parameter))
    {
        inventoryscreenextension.Add(() => InternalCreate(controltype, parameter));
        return true;
    }
    return false;
}
```

Die Check Methode Überprüft ob das Control sicher verwendet werden kann.

```
args = null;

if (controltype.IsAbstract)
    return false;

if (!typeof(Control).IsAssignableFrom(controltype))
    return false;

args = new object[inputargs.Length + 1];
args[0] = this;
Array.Copy(inputargs, 0, args, 1, inputargs.Length);

return true;
```

Für das DebugControl ändert sich die Signatur und Handhabung es können nur Nachrichten bzw. eine Func<string> registriert werden die den Text eines Labels Updated.

```
public bool RegisterOnDebugScreen(bool right, Func<string> updatefunc)
{
    if (updatefunc != null)
    {
        debugscreenextensions.Add((right, updatefunc));
        return true;
    }
    return false;
}
```

Das DebugControl Implementiert außerdem noch eine Methode um die obere Funktionalität umzusetzen. Es kann zwischen dem Linken und Rechten Panel gewählt werden.

```
    foreach ((bool right, Func<string> updatefunc) in
              screenManager.DebugControlExtensions)
         Register(right, updatefunc);

  public void Register(bool right, Func<string> updatefunction)
  {
      Label label = new Label(ScreenManager);
      if (right)
      {
          rightView.Controls.Add(label);
      }
      else
      {
          leftView.Controls.Add(label);
      }
      updatefunctions.Add(() => label.Text = updatefunction());
  }
```

Außerdem erfolgt das Update in einem try, catch Block und die fehlerhafte Meldung wird aus den Updatefunktionen ausgeschlossen wenn sie eine Exception verursacht.

```
int index = 0;
try
{
    for (; index < updatefunctions.Count; index++)
    {
        updatefunctions[index].Invoke();
    }
}
catch (Exception)
{
    //TODO: loggen
    if (index < updatefunctions.Count && index >= 0)
        updatefunctions.RemoveAt(index);
}
```

Alle anderen Änderungen in den Files beziehen sich auf Anpassungen die aufgrund der Einführung der neuen Interfaces notwendig waren. Sowie kleine generelle clean ups. Die Logik des Programms wurd nicht wesentlich verändert und es verhält sich immer noch so wie zuvor (abgesehen von der Physik…). Das Komponenten System benötigt immer noch eine Refaktorisierung ich hoffe ich kann mich dem nach den Definitions widmen. Das ISerializable Interface bekommt einen IGameService Parameter und nicht den IDefinitionManager. Implementierung der Entitykollision sowie Entity – Player Interaktion folgt darauf. Rechtschreibfehler und sonstiges darf behalten werden Sowie verweise auf Code Style ☺