Änderungen:

Renderthread

* Renderer Loader asynchron
  + Nutzung einer Klasse DoubleBufferedAsyncQueue, die auszuführende Aktionen enthält
  + Auslagerung der ModelGenerierung in ModelFactory
  + Auslagerung der Übertragung von Daten in die Klassen VAOTransmitter, TextureTransmitter und DataTransmitter
  + Implementiert noch nicht DoubleBuffer, sollte das aber tun, genauso, wie DoubleBufferedAsyncQueue. Das Interface kam aber erst danach.
* MasterRenderer
  + verpackt Renderable Inhalt
  + noch nicht vollständig
    - Model nur Referenz (aktuell gut, da Model sich nachher noch ändert, um anzuzeigen, dass es geladen ist)
      * Loader sollte so umgebaut werden, dass er nur noch synchron ist, dann kann die isLoaded Struktur in Model wieder entfernt werden
    - Materials ist nur Referenz auf Array -> auch die Materials können im Nachgang noch geändert werden, was aktuell aber nicht zu Problemen führt, da das Array nur in draw() genutzt wird. Kann aber zu Inkonsistenz führen.
      * Inhalt der Materials selbst kann auch noch nebenläufig geändert werden -> Gefahr einer Race Condition, da das Material im Renderthread geladen wird.
      * Materials sollten irgendwie final sein
    - JointMatrices ist nur Referenz auf Array -> Animationen können Race Conditions haben. Auch hier müssten die Matrizen kopiert werden, könnte durch automatisch vergrößernde Arrays gelöst werden. Problem, die meisten Renderables besitzen keine Joints -> evtl zwei verschiedene Caches?
    - TransformationsMatrix wird vollständig kopiert (effektiv final)
    - World Position wird vollständig kopiert (effektiv final)
  + Cache zur Generierung und Wiederverwendung der Objekte
    - Könnte in eigene Klasse ausgelagert werden
    - Double Buffer Konzept
  + Asnychrones setzen der Uniforms mit DoubleBuffer und Runnables
    - Runnable könnte vermieden werden, wenn Global Uniforms nicht mehr statisch ist, sondern Objekte davon erzeugt werden könnten.
  + Refactoring: Auslagerung von OpaqueMasterRenderer und TransparentMasterRenderer, weil grundlegend unterschiedliche Aufgaben durchgeführt werden.
  + Renderer ist nun DoubleBuffer
    - Evtl architektonisch nicht ganz schön, da Renderer damit eigentlich nichts zu tun hat, Trennung evtl besser als Vererbung, da potenziell nicht jeder Renderer einen DoubleBuffer benötigt. -> Unterscheidung in ConcurrentRenderer, der zusätzlich DoubleBuffer enthält?
* OpaqueMasterRenderer und TransparentMasterRenderer
  + DoubleBuffer für Renderables
    - Bei Clear() gehen Elemente der Tiefen Struktur verloren
    - Möglichkeiten suchen, um tiefe Struktur aufzulösen?
    - Buffer ist durch ReadWriteLock geschützt, das sollte unnötig sein, da swapBuffer sequentialisiert aufgerufen wird. Leider gibt es keine Möglichkeit das zu erzwingen. SwapBuffer könnte so auch nebenläufig aufgerufen werden, was aber nicht sinnvoll ist, da der Buffer zwischen Framewechseln getausch werden soll. Ansonsten entsteht die Gefahr, dass Renderables in dem falschen Frame gerendert werden.
  + Generatoren für Collectionelemente ausgelagert
    - Interface Constructor mit create() Methode -> generisches Factory Interface, evtl in Factory umbenennen, um Vertrautheit zu nutzen
* UiRenderer
  + besitzt keine Renderliste sondern nutzt den UiBaum, um an die enthalten UiElemente zu gelangen (status quo)
  + DoubleBuffer im Renderer ist deshalb sehr schwierig, da die gesamte UI Struktur gebuffert werden müsste, was zu Konsistenzproblemen führt
  + -> UIItem
    - Nutzung eines speziellen Renderables DoubleBufferedUIRenderable
      * Enthält zwei UIRenderables, die jeweils geklonte Materials nutzen. Das ist wichtig, weil die UI die Eigenschaften des Materials nutzt, um Änderungen zu realisieren
        + Text wird gerendert, indem auf der CPU eine Textur erzeugt wird, die dann an die Grafikkarte übergeben wird und als Textur für den Quad genutzt wird
        + Die Farbe des Textes wird über das Attribut tint der Textur angepasst
      * DoubleBuffer kopiert nur
        + Textur
        + Tint
        + Transformationsmatrix
      * UIRenderables besitzen nur ein Material, daher ist das ausreichend
      * Zudem ändert sich das Model der UIRenderables nie, es ist immer das statisch definierte GUIQuad. Die Form wird über Transformation bestimmt.
* RawModel besitzt eine feinere isLoaded Struktur
  + Implementiert AsyncGraphicsObject -> kann entfernt werden, wenn Loader Callbacks nutzt.
* RenderThread Klasse
  + Kapselt Aktionen, die vom RenderThread durchgeführt werden.
    - loadGraphics muss synchron durchgeführt werden
    - Möglichkeit zur Änderung des SwapIntervalls während der Programmlaufzeit
    - submitRender führt das Rendering und den Bufferswap des Bildes aus
* GameLoop
  + Extrahierung einer Klasse GameLoop
  + Nutzung von Runnables und Suppliers, um die Konfiguration als Server oder Client/Einzelspieler ohne Code-Doppelung zu ermöglichen.
  + Aufteilung in einen SimulationsStep und einen WindowStep, der WindowStep führt auch den SimulationsStep aus.
  + Alle Elemente, die sich auf das Rendering beziehen, sind in stepWindow() enthalten.
  + Der neue GameLoop ist damit
    - loadGraphics()
    - processInput
    - submitSimulation
    - SubmitRender
    - Join

Jobsystem

* BlocklibExecutor
  + Implementiert BlocklibExecutorService
  + Delegiert die Ausführung von Tasks and zwei ScheduledThreadpoolExecutors abhängig von der TaskPriority
  + Delegiert die Verpackung von Tasks, sodass diese ScheduledCompletableFutures sind
  + Zählt die Anzahl der aufgegebenen Tasks anhand ihrer Klasse für Statistik/Debugging-Zwecke
* CompletableFutureWrapper
  + Erzeugt CompletingCallables aus Callables und Runnables
  + Nutzt DelegatingCompletabelFutures, da die darunterliegenden ExecutorServices eigene Futures erzeugen, die für die Interaktion genutzt werden müssen.
* CompletingCallable
  + Erhält ein CompletableFutureObjekt und ein Callable
  + Führt das Callable aus und ruft entsprechend complete oder completeExe im CompletableFuture auf.
* RunnableCallable
  + Verpackt ein Runnable als Callable<Void> oder mit einem Rückgabewert
* DelegatingCompletableFuture
  + CompletableFuture, das ein inneres Future nutzt und Methodenaufrufe an dieses weitergibt.
  + Besitzt eine setMethode, da die Erzeugung des inneren Future-Objekts später geschehen muss.
* NamedThreadFactory
  + Erzeugt Threads mit bestimmten Namesn Präfixen, um das Debugging zu vereinfachen
* TaskPriority
  + Enum, das die Priorität eines Tasks definiert, aktuell sind nur die Prioritäten NORMAL und BACKGROUND vorhanden. Sobald der Executorservice so umgebaut ist, dass er auch Prioritäten unterstützt, können mehr Prioritäten genutzt werden.

Konsolidierung:

* Alle Stellen, die bisher schon Threads genutzt haben, werden so angepasst, dass sie nun einen BlocklibExecutorService nutzen
* Übergabe geschieht mittels Konstruktor
* Context wird der Service mittels Konstruktor übergeben und gibt ihn an Manager weiter
* EventManager
  + benachrichtigt nun mittels BlocklibExecutorService sequentiell. Muss aktuell sequentiell sein, da die Eventlistener nicht darauf ausgelegt sind, dass Events parallel ausgegeben werden.
* Fluid
  + Nutzt BlocklibExecutor um in Regelmäßigen Abständen den neuen Zustand zu berechnen -> Keine Notwendigkeit für Thread.sleep mehr. Stattdessen scheduleAtFixedRate(…)
* ChunkStorage und abgeleitete Klassen
  + loadChunk. saveChunk, doLoadChunk und meshChunk nutzen Executor mit TaskPriority.BACKGROUND, damit die Berechnung des nächsten Bilds nicht aufgehalten wird.
* Nicht Konsolidiert:
  + Aus Zeitgründen:
    - NioClientNetworkLayer
    - NioServerNetworkLayer
  + Weil Dependency Injection kompliziert ist
    - ConnectionInfo