Architektur Kronos

Im Folgenden ist die Architektur von Kronos beschrieben.

Architektur des Globus

Im Mittelpunkt der Globus-Komponente von Kronos steht die Klasse `Globe`. Eine Instanz dieser Klasse wird Client-seitig von der Kronos-View erzeugt. Der Globe hält intern einen `ResourcePool` mit einer fixierten Menge an `GlobeTiles`, die für das eigentliche Rendering des Globus verantwortlich sind. `GlobeTile` ist eine Klasse, die ein Quadrat im Lat/Long-Koordinatensystem repräsentiert. Die Seitenlänge dieses Quadrats hängt von der Zoomstufe des Globus ab:

Zoomstufe 0 teilt den Globus in 2x1 Tiles der Größe 180°x180° auf.
Zoomstufe 1 teilt den Globus in 4x2 Tiles der Größe 90°x90°.
Zoomstufe 2 teilt den Globus in 8x4 Tiles der Größe 45°x45°.
Zoomstufe n teilt den Globus in 2^{n+1}x2^{n} Tiles der Seitenlänge 180 * 2^{-n}.

Standardmäßig ist die Zoomstufe auf Werte zwischen 0 und 5 inklusive beschränkt und ändert sich mit der Kameraposition: je näher die Kamera am Globus liegt, desto höher die Zoomstufe. Dieses Zoomverhalten kann in der Konfigurationsdatei verändert werden.

Jedes `GlobeTile` beinhaltet OpenGL- und VTK-Ressourcen, die zur Darstellung des Tiles benötigt werden: Textur, Shader und Actor. Die Vertexdaten werden in Abhängigkeit der Differenz der maximalen und minimalen Höhe des Terrains innerhalb eines `GlobeTiles` aus unterschiedlich detaillierten Plane-Meshes bezogen.

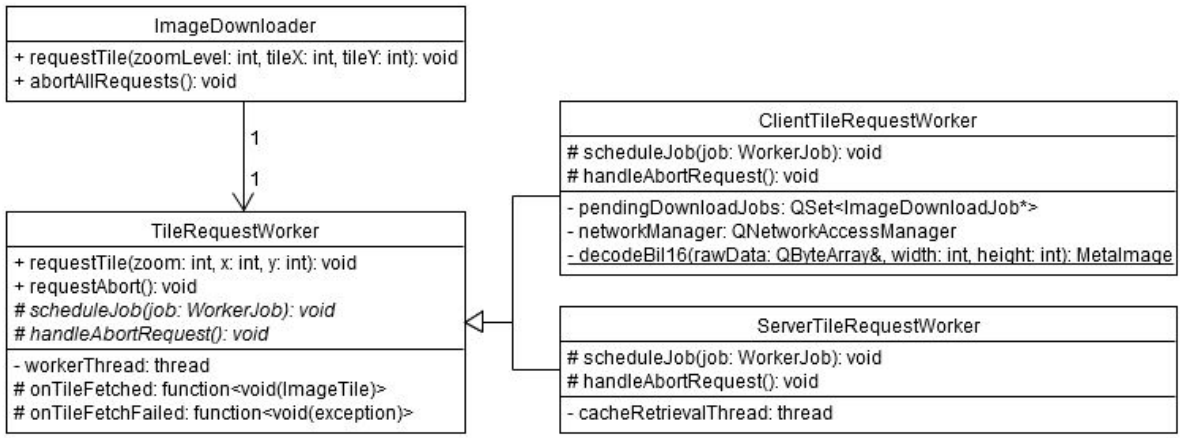
Anfangs sind alle Tiles inaktiv, d. h. sie besitzen ein inaktives oder abgelaufenes Handle zum `GlobeTile-ResourcePool`. Der `ResourcePool` verwaltet Instanzen einer Klasse bis zu einer fixierten Maximalanzahl. Instanzen können deaktiviert und aktiviert werden. Ist die Maximalzahl an Instanzen erreicht und wird eine neue Instanz angefordert, wird die älteste deaktivierte Instanz recyclet und als neue Instanz zurückgegeben, wobei Handles, die auf diese recyclete Instanz zeigten, als abgelaufen markiert werden. Ansonsten wird eine frische Instanz erzeugt. Nicht-abgelaufene, deaktivierte Instanzen können reaktiviert werden und behalten ihre Daten bei. Abgelaufene Instanzen müssen neu angefordert und initialisiert werden.

Bei jeder Kamerabewegung wird im Globe geprüft, welche `GlobeTiles` momentan sichtbar sind. `GlobeTiles`, die sich von der Kameraperspektive aus gesehen auf der Rückseite des Globus befinden, werden zuerst ausgeblendet. Sollte das Tile noch sichtbar sein, wird danach geprüft, ob das Frustum der Kamera die Bounding Box des `GlobeTiles` schneidet. Ist dies der Fall, wird das `GlobeTile` aktiviert und angezeigt. Wurde ein recycletes oder frisches Tile aus dem `ResourcePool` bezogen, werden die Tile-Daten geladen. Dafür wird eine asynchrone Anfrage an den `TileDownloader` gestellt, das entsprechende Tile von den

NASA-Servern oder vom lokalen Cache zu laden. Dem Tile wird währenddessen eine temporäre Ladetextur zugewiesen. Wurde das Tile geladen, wird eine Callback-Funktion aufgerufen und das `GlobeTile` erhält eine kombinierte RGB- und Heightmap-Textur. Heightmap-Daten werden im Alphakanal der Textur kodiert, um später im Vertex-Shader zu Höhenverschiebungen verarbeitet zu werden.

Tile Download

Zu den Aufgaben des Tile Downloaders zählen der Download der Satellitenbilder und Heightmaps für den Globus von den NASA Servern, sowie das Ablegen und Laden der bereits heruntergeladenen Texturen in einem Cache.
Um Tiles anzufragen, wird eine `ImageDownloader` Instanz benötigt. Diese stellt Methoden zum Anfragen von `Tiles`, sowie dem Abbruch bereits laufender Tile-Anfragen bereit.



Die wichtigsten am `TileDownload` Prozess beteiligte Klassen und Funktionen

Tile Requests

Für jede Tile Anfrage wird ein neues `TileRequest` Objekt erstellt, dass die Zoomstufe, sowie X und Y Koordinaten des angefragten Tiles enthält (Dabei bezeichnet die Koordinate `(0, 0)` Das oberste linke Tile und `((2^(zoom+1))-1, (2^zoom)-1)` das unterste rechte Tile für eine gültige Zoomstufe >= 0). Das Anfrageobjekt wird in eine `requestQueue` eingefügt. Diese Queue wird asynchron in einem Worker Thread regelmäßig auf neue Einträge überprüft. Wurde eine ausstehende Tile Anfrage in der Schlange gefunden, wird zunächst im Cache nach dem entsprechenden Tile gesucht. Wurden alle für das Tile relevanten Ebenen gefunden (Textur und Heightmap), wird ein entsprechendes `ImageTile` Objekt erstellt, das Bilder für alle Ebenen, die X und Y Korrdinaten, sowie die Zoomstufe des angefragten Tiles enthält. Das `ImageTile` wird dann durch Aufrufen des `onTileFetched` Callbacks an den Globus übergeben.
Wurde mindestens eine Ebene nicht im Cache gefunden, so wird die Anfrage an eine Unterklasse des `TileRequestWorker`'s weitergeleitet. Insgesamt erben zwei Klassen vom `TileRequestWorker`. Diese sind auf die beiden verschiedenen Umgebungen angepasst, in denen Kronos ausgeführt werden kann.

ClientTileRequestWorker

Auf einem ParaView Client ist der `ClientTileRequestWorker` dafür verantwortlich, Downloads für alle nicht im Cache befindlichen Tiles zu starten. Dazu verwaltet auch er eine Job-Schlange, in der Anfragen für nicht gefundene Tiles abgelegt werden, sowie eine Menge an Jobs, die gerade in Arbeit sind. Für jede Anfrage werden Downloads für alle fehlenden Ebenen gestartet. Heruntergeladene Satellitenbilder werden direkt im Cache abgelegt und in einem dem jeweiligen Job zugeordneten `ImageTile` abgelegt, Heightmaps werden von den NASA Servern als Binärdaten im Bil16 Format bereitgestellt und müssen erst decodiert werden, bevor sie als Bilder im Cache gespeichert werden.
Sind alle einem Job zugeordneten Downloads abgeschlossen und haben keine Fehler verursacht (z.B. ungültige Antworten vom Server oder Timeouts), wird das dem Job zugeordnete `ImageTile` durch einen Aufruf des `onTileFetched` Callbacks an den Globus weitergereicht.

ServerTileRequestWorker

Auf ParaView Servern kommt ein `ServerTileRequestWorker` zum Einsatz. Er reiht Anfragen für unvollständige Tiles ebenfalls in eine Schlange ein, allerdings startet er niemals selbst Downloads, sondern wartet so lange, bis alle angefragten Tile Ebenen im Cache erscheinen. Sobald das Tile komplett ist, wird ein entsprechendes `ImageTile` über das `onTileFetched` Callback an den Globus geschickt. Damit der `ServerTileRequestWorker` funktioniert, muss der Cache zwischen allen ParaView Instanzen durch externe Mittel synchronisiert werden, beispielsweise ein gemeinsames Netzwerklaufwerk.

Abort Requests

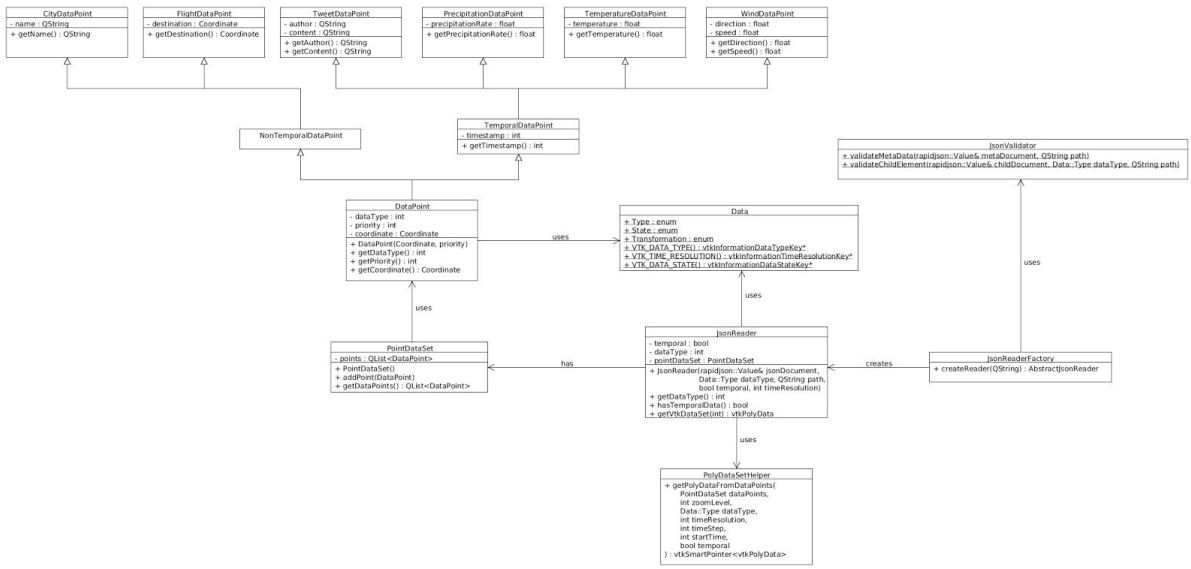
Abort Requests werden vor allem bei Zoomstufenwechseln gestellt. Wird die `requestAbort()` Methode des `TileRequestWorker`'s aufgerufen, wird zunächst die `requestQueue` des `TileRequestWorker`s geleert und eine neue `AbortRequest` in selbig eingereiht. Die beiden Subklassen reagieren darauf dann im Falle eines `ClientTileRequestWorker`'s durch einen Abbruch aller laufenden Downloads, beziehungsweise beim `ServerTileRequestWorker` durch das Löschen aller ausstehenden Tile Anfragen.

Fehlerbehandlung

Treten beim Download oder dem Laden von Bildern aus dem Cache Fehler auf, wird eine enstprechende Exception über das `onTileFetchFailed` Callback weitergereicht. Dort kann dann entsprechend auf den Fehler reagiert werden (z.B. durch Neustart des Downloads oder Ähnliches).

Reader

Die Klassenstruktur im Ordner `src/Reader` ist für das Einlesen der `kJson`-Daten zuständig, deren Struktur im Handbuch beschrieben wurde.



KronosReader

In der obersten Ebene befindet sich `vtkKronosReader.cxx`. Diese Klasse stellt die Schnittstelle zu VTK dar, übernimmt die Fehlerbehandlung und erstellt mit dem gegebenen Dateipfad und -name einen neuen `JsonReader`. An diesen wird bei Datenanfragen die angefragte Zeit und das aus der Kameraposition berechnete Zoomlevel für das Level-of-Detail-System weitergereicht.

Außerdem fügt der `KronosReader` allen Daten, die in die Pipeline weitergereicht werden, die richtigen Metainformationen hinzu, die von den Filtern später benötigt werden.

Nach Erstellen eines `KronosReader` beginnt dieser, alle Daten in einem Hintergrund-Thread für den späteren schnelleren Zugriff zu cachen.

JsonReader

Die Klasse in `JsonReader.cpp` übernimmt das eigentliche Lesen der Daten. Ein solcher Reader muss über die `JsonReaderFactory` erstellt werden. Die erkennt die Metainformationen der Datei wie Datentyp, zeitliche Auflösung und ob es sich um temporale Daten handelt. Zudem prüft die Factory die JSON-Datei auf syntaktische Validität und schließlich mithilfe des `JsonValidator` auch auf semantische Validität, also ob die Struktur der Daten korrekt ist. Ist dies nicht der Fall, wird eine passende Exception geworfen, die der `KronosReader` fängt und an den Nutzer weiterreicht.

Ist der JsonReader erstellt, so liest er zunächst alle Datenpunkte aus der JSON-Datei unter Verwendung des rapidJson-Library aus und schreibt sie in eine interne Repräsentation in ein `PointDataSet`, die im nächsten Unterpunkt thematisiert wird.

Nun stellt der `JsonReader` eine Schnittstelle zur Verfügung, um von den gelesenen Daten `vtkPolyData`-Objekte für ein bestimmtes Zoomlevel und einen bestimmten Zeitschritt zu generieren. Dazu werden die Punktdaten an den `PolyDataSetHelper` weitergereicht, der die Konvertierung übernimmt.

Der `JsonReader` schreibt per Default angefragte Daten in einen Cache, um einen späteren erneuten Zugriff zu beschleunigen.

Datenrepräsentation

Auf oberster Ebene existiert zunächst eine Klasse `Data`, die einige Enums und Methoden enthält. Diese hängen mit den Metainformationen zusammen, die den Daten in der Pipeline angehängt werden. Um das Propagieren dieser Metainformationen möglich zu machen, mussten deren Schlüsselwerte im Ordner `MetaInformationKeys` neu implementiert werden.

An der Spitze der eigentlichen Klassenhierarchie steht der `DataPoint`. Er ist am allgemeinsten und enthält nur den Datentyp (also `CITIES`, `FLIGHTS`, `TWEETS`, `PRECIPITATION`, `TEMPERATURE`, `WIND` oder `CLOUD_COVERAGE`), die Koordinaten und das Zoomlevel eines Datenpunkts.

Von ihm erben `NonTemporalDataPoint` und `TemporalDataPoint`. Letzterer ergänzt den eigentlichen Datenpunkt um einen Timestamp.

Von `NonTemporalDataPoint` erben schließlich `CityDataPoint` und `FlightDataPoint`, die die jeweiligen eigentlichen Daten wie Städte- oder Airlinenamen speichern.

Von `TemporalDataPoint` erben `CloudCoverageDataPoint`, `PrecipitationDataPoint`, `TemperatureDataPoint`, `TweetDataPoint` und `WindDataPoint`, wieder mit ihren eigentlichen Daten.

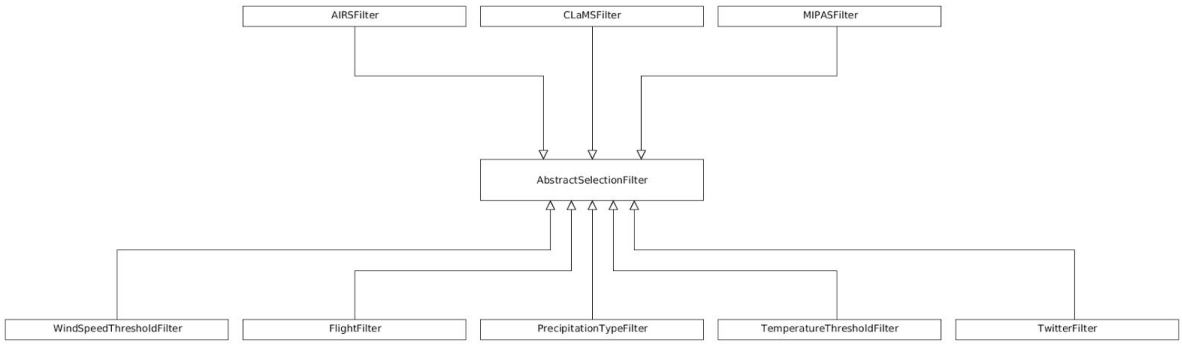
Filter

Unsere Filter befinden sich im Order `STUPRO/src/Filter` und werden aufgrund unserer CMake-Konfiguration automatisch kompiliert und in das Projekt eingebunden. Alle Filter sind in der von ParaView geforderten XML-Datei beschrieben (`KronosFilter.xml`). Um die Filter in ParaView in verschiedenen Kategorien zu gruppieren, haben wir diese in der XML-Datei den Kategorien `Kronos` und `SciVis` zugewiesen. Ansonsten werden in der XML-Datei die

Elemente, welche in der Benutzeroberfläche angezeigt werden, beschrieben und mit den entsprechenden Codeteilen verknüpft.

Über Metainformationen bezüglich des Datentyps, -zustands und -transformationszustands, die Daten anhaften während sie die Pipeline durchqueren, können Reader und Filter miteinander kommunizieren, indem sie die Datenmetainformationen passend ändern oder die Daten auf Kompatibilität mit dem jeweiligen Filter testen.

Selektionsfilter



Alle Filter, die Daten aufgrund von bestimmten Attributen filtern, erben von der von uns erstellten abstrakten Superklasse `AbstractSelectionFilter`, welche die Grundfunktionen des Filters festlegt. Sie implementiert die von ParaView für einen Filter geforderten Methoden wie `FillOutputPortInformation` oder definiert den Umgang mit Fehlern. Die Kindklassen wie `FlightFilter` oder `TwitterFilter` müssen nun nur noch die kompatiblen Datentypen angeben und eine Funktion namens `evaluatePoint` implementieren, welche per zurückgegebenem Boolean angibt, ob ein bestimmter Datenpunkt herausgefiltert wird oder nicht. Diese Methode wird dann für jeden Datenpunkt in der obligatorischen Methode `RequestData` der Superklasse aufgerufen, wobei alle sichtbaren Punkte und deren zugehörige Daten-Arrays entsprechend ihrer Struktur in die Ausgabe des Filters geschrieben werden.

Einige der Filter beruhen darauf, dass eine gewisse Datengrenze in Form von Minimal- und Maximalwerten angegeben wird. Hierfür wird ein spezielles ParaView-Widget verwendet, dass genau dafür zwei Slider darstellt und deren minimal und maximal einstellbaren Werte von den Minimal- und Maximalwerten in den Daten abhängig macht.

Im Folgenden werden alle Filter beschrieben, die einen solchen Selektionsfilter implementieren.

AIRS-Filter

Der Filter in `AIRSFilter.cpp` kann AIRS-Daten ausgehend von minimalen und maximalen Werten der Zeit, des Schwefeldioxidindex' und des Ascheindex' filtern.

CLaMS-Filter

Der Filter in `CLaMSFilter.cpp` kann CLaMS-Daten ausgehend von minimalen und maximalen Werten der Zeit, der Höhe, der Temperatur, des Drucks, der Wirbelstärke und der potentiellen Temperatur filtern.

MIPAS-Filter

Der Filter in `MIPASFilter.cpp` kann MIPAS-Daten ausgehend von minimalen und maximalen Werten der Zeit, der Höhe, des Orbitindex', des Detektionsindex' und des Profilindex' filtern.

Flug-Filter

Der Filter in `FlightFilter.cpp` kann Flugdaten ausgehend von Airline-Name, Flughafenkürzel von Start- und Zielflughafen und Fluglänge filtern. Dabei ist es möglich nach exakten oder enthaltenen Suchbegriffen im Airline-Name zu filtern.

Niederschlagstyp-Filter

Der Filter in `PrecipitationTypeFilter.cpp` kann Niederschlagsdaten ausgehend von dem Niederschlagstyp der Punkte filtern, welcher entweder Regen, Schnee, Schneeregen, Hagel oder undefiniert ist.

Temperaturbereich-Filter

Der Filter in `TemperatureThresholdFilter.cpp` kann Temperaturdaten ausgehend von einem Bereich aus Minimal- und Maximaltemperatur filtern.

Twitter-Filter

Der Filter in `TwitterFilter.cpp` kann Twitterdaten ausgehend von Autoren, einem Bereich aus Minimal- und Maximalanzahl der Retweets und nach Suchbegriffen im Text des Tweets filtern. Dabei ist es möglich nach exakten oder enthaltenen Suchbegriffen im Autorenname zu filtern.

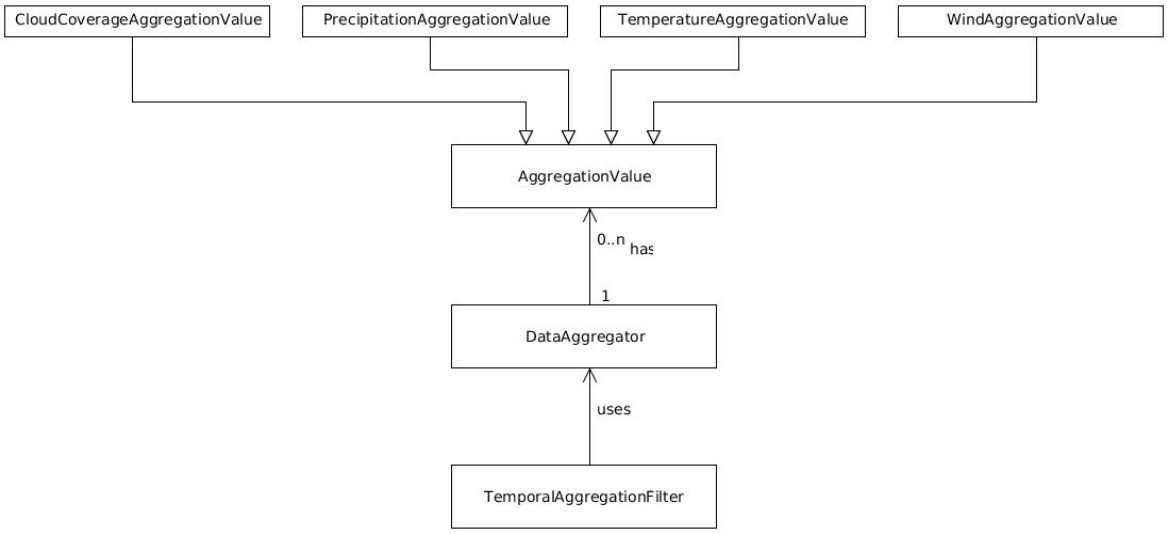
Windgeschwindigkeitsbereich-Filter

Der Filter in `WindSpeedThresholdFilter.cpp` kann Winddaten ausgehend von einem Bereich aus Minimal- und Maximalwindgeschwindigkeit filtern.

Zeitliche Filter

Kronos enthält zusätzlich zwei Filter, die über mehrere Zeitschritte hinweg Daten verarbeiten.

Zeitlicher Aggregationsfilter

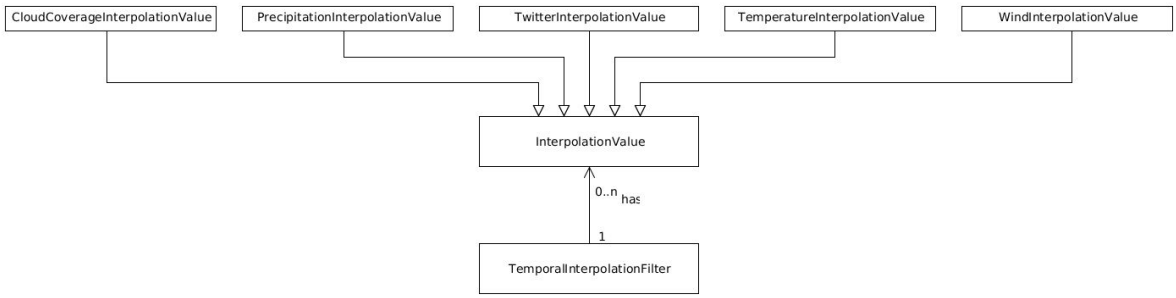


Der Filter in `TemporalAggregationFilter.cpp` kann zeitliche Daten aggregieren. Hierbei werden Temperaturen, Windgeschwindigkeiten und -richtungen sowie Bewölkungsgrade gemittelt. Niederschlagsraten werden zu einer gesamten Niederschlagsmenge akkumuliert.

Der Filter kann mit dem Fehlen eines Datenpunkts zu einem Zeitpunkt umgehen. Bei Niederschlagsraten wird hierfür angenommen, dass die Niederschlagsrate konstant blieb. Sollte lieber eine lineare Interpolation der Werte stattfinden, kann vor der Aggregation der unten beschriebene zeitliche Interpolationsfilter verwendet werden.

Der Filter muss hierfür den Typ der Eingabedaten sowie die Anzahl der Sekunden in einem Zeitschritt kennen, welche als Metainformationen ankommen. Die eigentliche Arbeit findet im `DataAggregator` statt, der die Daten je nach Typ korrekt aggregiert und in einer internen Repräsentation ablegt. Für die Mittelung von Werten wird hierbei das Cumulative Moving Average verwendet, um sehr große Summen zu vermeiden. Zuletzt kann der `DataAggregator` die aggregierten Daten als `vtkPolyData` ausgeben.

Zeitlicher Interpolationsfilter



Der Filter in `TemporalInterpolationFilter.cpp` kann zeitliche Daten interpolieren. Im Gegensatz zum VTK-eigenen zeitlichen Interpolationsfilter kann er auch mit dem Fehlen eines Datenpunkts zu einem Zeitpunkt umgehen und cacht Daten. Für die Interpolation von Daten legt der Filter diese in einer internen Repräsentation ab und erzeugt nach vollendeter Arbeit ein `vtkPolyData`-Objekt daraus.

Der Filter kann mit Bewölkungsgraden, Niederschlagsdaten, Temperaturwerten, Winddaten und Twitterdaten, auf die der `Density Heatmap Filter` angewandt wurde, umgehen.

Der Filter arbeitet in zwei Stufen um die Performance durch Daten-Caching zu erhöhen. Bei einer Anfrage von Daten zu einem bestimmten Zeitpunkt wird zunächst überprüft, ob ein Cache existiert. Ist dies nicht der Fall wird zunächst die interne Repräsentation aufgebaut, indem durch alle Zeitschritte iteriert wird. Danach wird der erste und letzte Zeitschritt mit allen vorkommenden Punktkoordinaten und den als bis dahin konstant angenommenen Werten gefüllt. Alle fehlenden Werte zu bestimmten Zeitschritten werden schließlich mittels linearer Interpolation aufgefüllt.

Ist der Cache aufgebaut, so wird zwischen den beiden ganzzahligen Zeitschritten vor und nach dem angefragten Zeitschritt linear interpoliert und das Ergebnis als `vtkPolyData`-Objekt ausgegeben.

Geographische Filter

Die folgenden Filter agieren auf den geographischen Grundinformationen der eingelesenen Daten.

Geodäten-Generierungs-Filter

Der Filter in `GenerateGeodesics.cpp` kann ausgehend von Flugdaten die Geodäten der Flüge generieren. Dabei kann der Detailgrad und die Höhe der Kurve eingestellt werden. Die

maximale Rekursionstiefe beschraenkt hierbei die Anzahl der Ausfuehrungen einer `while`-Schleife.

Globus-Transformations-Filter

Der Filter in `SphericalToCartesianFilter.cpp` kann GPS-Daten (Breiten-, Längengrad und Höhe) in kartesische Koordinaten konvertieren. Er kann genutzt werden, um flache Daten auf dem Globus darzustellen.

Daten-Überhöhungs-Filter

Der Filter in `TerrainHeightFilter.cpp` nutzt statische Höhendaten des Terrains, um Datenpunkte entsprechend eines Überhöhungsfaktors auf die richtige Höhe zu bringen. Dazu werden die Hoeheninformationen aus der Heightmap ausgelesen.

Weitere Filter

Datendichte-Heatmap-Filter

Der Filter in `HeatmapDensityFilter.cpp` kann genutzt werden, um die Dichte von Datenpunkten (z.B. Tweets) zu visualisieren. Dazu kann im Filter eine vertikale und horizontale Auflösung eingestellt werden, die genutzt wird, um über den Eingangsdaten ein gleichmäßiges Punktgitter aufzuspannen. Jeder Datenpunkt der Ausgangsdaten wird dann dem Gitterpunkt zugeordnet, dem er am nächsten ist. Dessen Datendichteindex wird für jeden zugeordneten Punkt erhöht.

Schließlich wird das Punktgitter mit den Dichtewerten als `vtkPolyData`-Objekt ausgegeben.

Wind-Beschleunigungsvektor-Berechnungs-Filter

Der Filter in `WindVelocityVectorCalculationFilter.cpp` wird für die Strömungsvisualisierung benötigt. Er berechnet aus Windrichtung in Grad und Windstärke für jeden Punkt einen Beschleunigungsvektor und schreibt diese in ein neues Array, welches samt der Ausgangsdaten ausgegeben wird.

Datendichte-Reduktions-Filter

Der Filter in `DataDensityFilter.cpp` wird zur manuellen Verringerung der Dichte von Punktdaten eingesetzt, falls das in Kronos bereits auf Reader-Ebene unterstützte Level of Detail nicht ausreicht.

Der Filter fasst Datenpunkte ausgehend von einem einstellbaren Datenreduktionsgrad zusammen und mittelt deren numerische Werte. Die reduzierten Datenpunkte mit deren Mittelwerten werden schließlich als `vtkPolyData`-Objekt ausgegeben.

Die Zusammenfassung der Datenpunkte kann je nach Wunsch des Nutzers entweder mit einem simplen Algorithmus, der Punkte der Reihe nach mit naheliegenden Punkten zusammenfasst, oder mit dem intelligenteren aber langsameren kMeans-Clustering erfolgen.

Ersteres eignet sich eher für gleichförmige Datenpunkte, etwa in Gittermuster, letzteres kann gut für Daten eingesetzt werden, die in Clustern verteilt sind (z.B. Tweets in Städten).

Glossar

Begriff	Erklärung
Paraview-Client	Die ParaView Instanz, über die die gesamte Benutzerinteraktion erfolgt.
Paraview-Server	ParaView Instanz, die auf einem Server (Beispielsweise der Powerwall) läuft. Ein ParaView Client kann sich zum Visualisieren von Daten mit einem ParaView Server verbinden.
Cumulative Moving Average	Mittlungsalgorithmus zur Berechnung eines Durchschnittswerts, der sehr performat ist.
kMeans-Clustering	Der kMeans-Algorithmus partitioniert n Beobachtungen in k Cluster, wobei jede Beobachtung zu dem Cluster mit dem nächsten Mittel Gruppiert wird. Es handelt sich hierbei um einen iterativen Algorithmus. Er teilt den Datensatz in Voronoi Zellen.