Konzepte zur 3D-Visualisierung von Laserscanner-Daten mit VTK und Python

Florian EDELMAIER, Bernhard HÖFLE und Armin HELLER

Zusammenfassung

Die große Datendichte und die damit verbundene Datenmenge von Laserscanner-Daten übersteigen zumeist die Kapazitäten klassischer Softwaresysteme. Dynamische 3D-Visualisierungssysteme bedienen sich verstärkt verteilter Architekturen und des *Parallel Processing*, um diese Datenvolumina erfolgreich zu bewältigen.

In Kooperation mit der Forschungsgruppe Laserscanning des Institutes für Geographie der Universität Innsbruck und des alpS-Zentrums für Naturgefahren Management, die engagiert an einem *Open Source* Informationssystem für Laserscanner-Daten entwickelt (HÖFLE et al. 2005), wurde am Beispiel des Auslesevorganges von Laserscanner-Daten aus einer räumlichen Datenbank und deren Visualisierung versucht, diese Schritte softwaretechnisch nachzuvollziehen. Zentrales Element ist dabei die 3D-Visualisierungsbibliothek VTK (*Visualization Toolkit*).

1 VTK – das Werkzeug

Das *Visualization Toolkit* (VTK) ist eine *Open Source*, objektorientierte, portable C++ Klassenbibliothek für 3D–Computergrafik, Visualisierung und Bildbearbeitung (SCHROEDER et al. 2005). Grundsätzlich stützt sich VTK auf zwei abstrakte Konzepte (SCHROEDER et al. 2005):

- Das Grafikmodell (*grafics model*) abstrahiert als Zwischenlayer die betriebssystemeigene Grafikprogrammiersprache. Unabhängig von Plattform und Gerät werden in diesem Modell abstrakte grafische Konzepte zusammengefasst.
- Die Visualisierungspipeline (*visualization pipeline*) transformiert Information in grafische Daten (AVILA et al. 2004). Oder anders ausgedrückt: Die Visualisierungspipeline konstruiert geometrische Darstellungen, die dann durch die Grafikpipeline *gerendert* werden. Die Pipeline besteht aus Objekten, die Daten repräsentieren, Prozessobjekte, die auf Daten operieren, und die gekennzeichnete Richtung des *data flow*.

2 Implementierung

2.1 Parallele Ausführung

Die meiste Zeit verbringt ein Computer damit, nichts zu tun. Eine genauere Betrachtung der CPU-Auslastung auf dem Systemmonitor untermauert diese These. Nur ganz selten lassen

sich Spitzen mit hundert Prozent Auslastung registrieren, auch wenn mehrere Programme ausgeführt werden. Schon früh in der Computerentwicklung sah man die Notwendigkeit, diese Ressourcen anzuzapfen, indem man mehrere Programme gleichzeitig ablaufen ließ. Durch die Teilung der CPU-Aufmerksamkeit auf mehrere definierte Aufgaben wird weniger Kapazität durch das Abwarten auf externe Events verschwendet. Diese Technik wird üblicherweise als *Parallel Processing* bezeichnet, weil verschiedene Aufgaben scheinbar zusammen, überlappend oder parallel abgearbeitet werden (LUTZ 2001).

Ein Prozess ist ein Programm in Ausführung. Wenn ein Programm gestartet wird, richtet das Betriebssystem einen Prozess ein. Der Prozess erhält eine Prozessidentifikation und im Arbeitsspeicher des Computers einen nach außen abgeschotteten Bereich, in dem die momentanen Inhalte der Variablen und vieles andere abgespeichert sind (Prozessumgebung). Ein Prozess wird erzeugt, nimmt während seiner Lebenszeit unterschiedliche Zustände ein und stirbt dann irgendwann (WEIGEND 2005).

In diesem Fall liegen die Vorteile, in der Ausführung paralleler Eingabe/Ausgabeoperationen. Im Vergleich zu CPU-Geschwindigkeiten sind I/O-Operationen relativ langsam. Während Maschineninstruktionen Nanosekunden zur Ausführung benötigen, liegt die Dimension für beispielsweise Suchoperationen auf der Festplatte im Bereich von Millisekunden (MATLOFF & HSU 2005). Wartet man nun auf die Beendigung einer solchen Suche, verschwendet man CPU-Zeit, in der Millionen von Maschineninstruktionen ausgeführt werden könnten.

Vereinfacht gesagt, stellen Betriebssysteme zwei Möglichkeiten zur parallelen Programmierung zur Verfügung: Einerseits Prozesse, die in separaten logischen Adresseinheiten von einander geschützt laufen und andererseits *Threads*, die sich parallel denselben Adressraum teilen, ohne voneinander abgeschottet zu sein. Das Fehlen des gegenseitigen Schutzes und die Bereitstellung nur eines geteilten Adressraumes führen zu geringerem *Overhead* und zu schnellerer Kommunikation. Funktionell basieren beide auf den Diensten des zugrunde liegenden Betriebssystems, um Bits von Code gleichzeitig ablaufen zu lassen. In der Ausführung unterscheiden sie sich aber in ihren Schnittstellen, der Portierbarkeit und in der gegenseitigen Kommunikation (LUTZ 2001).

2.2 postvtk Version 0.1

In Anlehnung an die Kombination der verwendeten Komponenten PostGIS und VTK nennt sich das Python-Modul, das eigene Programmieransätze beinhaltet, postvtk. Dabei stellt es verschiedene Verfahren zum Auslesen von Punktdaten aus einer PostgreSQL-Datenbank mit PostGIS Erweiterung zur Verfügung, die dann relativ kompakt in die Visualisierungspipeline von VTK eingebunden werden können.

Die Hauptaufgabe des Auslesens der Punktgeometrien und deren Formatierung in VTK-Objekte erfolgt in den *_Transmitter-Klassen. Oberste Priorität im gesamten Modul war, Methodiken der parallelen Prozessierung zum Erstellen eines vtkSource-Objektes anzuwenden und zu bewerten. Neben dem sequentiellen Standardausleseverfahren (Simple_Transmitter) steht noch nach dem Prinzip des replizierten Arbeiters ein Aufgebot an *worker threads* (Threaded_Transmitter) zur Verfügung, um den Overhead bei I/O-Operationen zu dämpfen, da durch das Python *Threading*-Modul keine eigentliche Gleichzeitigkeit unterstützt wird. Als dritte Möglichkeit wurde mit dem Ziel, auf Multiprozessor-

Plattformen wahre Gleichzeitigkeit zu unterstützen, die Klasse Forked_Transmitter im Stile der Unix/Linux fork-Methode implementiert.

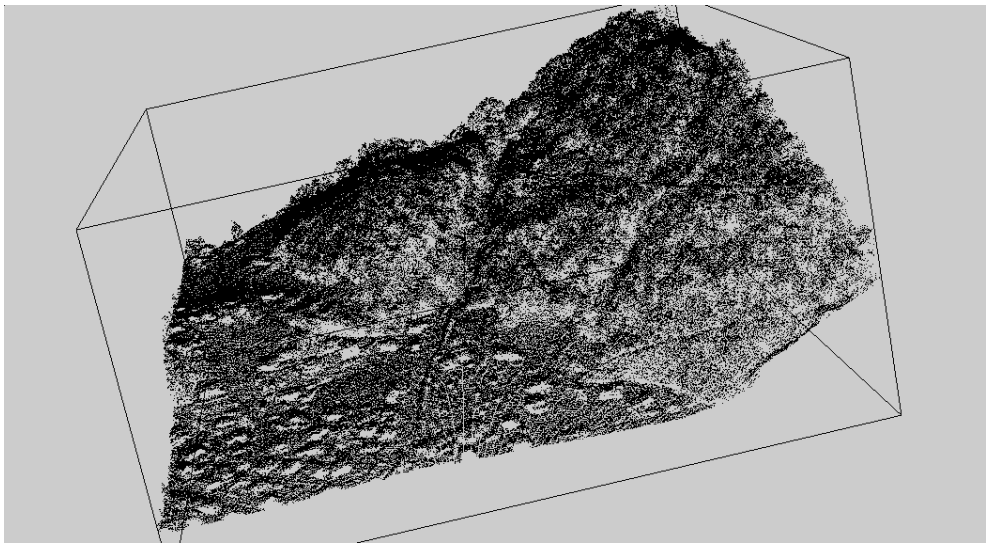

Abb. 1: Visualisierung eines Ausschnitts der Punktwolke mit VTK

Integraler Bestandteil der Transmitter-Familie ist ein *SQL-Statement*, das neben Spaltennamen und Tabelle weitere Ausleseeinschränkungen spezifiziert. Um der Komplexität gerecht zu werden, die neben benutzerfreundlicher Anwendung über formale Parameter auch die Logik zur Spaltung der Daten auf die einzelnen Prozesse beinhaltet, wurden eigene DQS-Klassen (*DatabaseQueryString*) für Transmitter-Klassen aggregiert. Die Basisklasse _DQS wird über ihre Subklassen mit einem *Dictionary*-Parameter, das über seine *key value* Paare Sucheinschränkungen hält, instanziert, um als einzige Aufgabe eine syntaktisch korrekte WHERE Klausel der SQL-Anweisung zu bilden. Die Tochterklasse Simple_DQS hat nun nur mehr die Aufgabe, die gesamte SQL-Anweisung zusammenzusetzen. Um gleichzeitig operierenden Prozessen bzw. Threads ihre eigenen Datenbereiche zuzuweisen, wird wiederum eine Subklasse Parallel_DQS von Simple_DQS gebildet, die als Rückgabewert mehrere SQL-Anweisungen liefert, mit denen dann ebenso viele Prozesse bzw. *Threads* angestoßen werden können. Da die Daten lokal auf einem Rechner und in einer Tabelle gespeichert sind, erschlossen sich nur zwei Möglichkeiten, die Abfragen zu teilen:

- Eine Aufteilung über eine festgelegte Anzahl von Werten, die jeder Prozess zu bearbeiten hätte, mittels einer LIMIT-Anweisung inklusive eines sich entsprechend ändernden <Start> Attributes.
- Eine Selektion der abgefragten Spalten, die pro Prozess auszulesen sind (d.h. jede einzelne Komponente der Geometrie (x,y,z)).

Letztere erwies sich nicht nur als performanter, sondern war auch eleganter in die bestehende Funktionalität einzubinden (für eine endgültige Bewertung siehe Kapitel 3).

3 Resümee

Integraler Bestandteil dieser Arbeit ist natürlich die Überprüfung, ob tatsächlich Verbesserungen im Ausleseverhalten durch Verwendung paralleler Methodik erzielt werden konnten. Zu diesem Zweck wurden alle drei implementierten Ausleseverfahren mit ansteigenden Datenvolumina auf unterschiedlichen Rechnern ausgeführt. Leider standen zu diesem Zweck keine Mehrprozessorrechner bzw. Rechner, auf denen die unterschiedlichen Prozessoren einzeln angesprochen werden konnten, zur Verfügung, was besonders die Bewertung der Klasse Forked_Transmitter erschwert. Die unterschiedlichen Diagramme zeigen aber entsprechend der Kapazität des einzelnen Rechners und der Anzahl der gemittelten Versuche einen ähnlichen Verlauf. Durch die Übereinstimmung der generellen Tendenzen wird nur ein repräsentatives Diagramm dargestellt (Abb. 2).

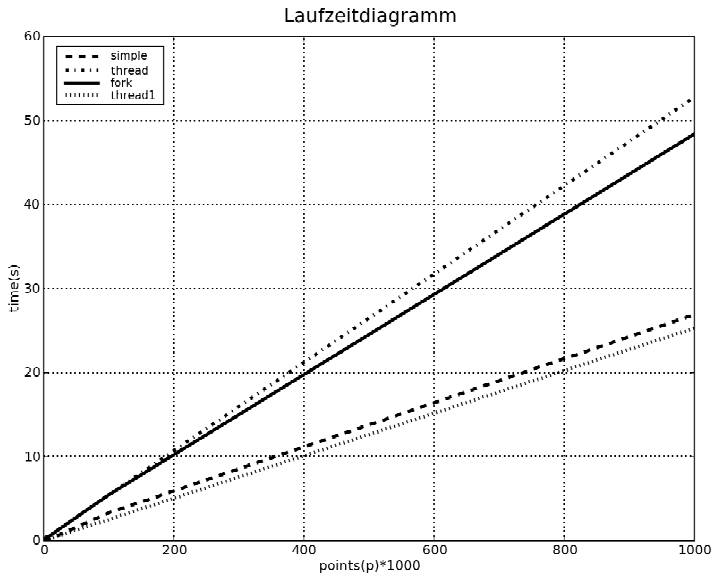

Abb. 2: Laufzeitmessungen

Die Laufzeiten beziehen sich immer auf den gesamten Visualisierungsvorgang – vom Einlesen der Daten bis hin zur fertigen Darstellung der Punktwolke. Da sich das eigentliche *Renderen* der VTK-Formate als konstant erwies, tut dies der Analyse des Auslesevorgangs keinen Abbruch.

Das Prozedere der Performancetests entspricht dabei folgendem Muster: Auslesen von 1.000, 10.000, 100.000 und von 1.000.000 Punkten durch die Klassen Simple_Transmitter, Threaded_Transmitter und Forked_Transmitter; Durchführung von jeweils 10 Versuchen und die Mittelung der einzelnen Laufzeiten.

Auf den ersten Blick scheint das Ergebnis ernüchternd: Das einfache sequentielle Ausleseverfahren durch die Klasse Simple_Transmitter ist eindeutig am schnellsten. War dieses

Ergebnis gleichsam „vorprogrammiert", liegt ein Fehler in der Logik oder ist es einfach so zu akzeptieren? Folgende Schlüsse und Verbesserungsmöglichkeiten ließen sich dabei ableiten:

- Die Tests beziehen sich zur Gänze auf Einprozessorsysteme, was für die Thematik eine unglückliche Situation darstellt. Dass sich aus der Instanzierung mehrerer *Threads* bzw. Prozesse ein zeitlicher *Overhead* ergibt, stellt grundsätzlich keine neue Erkenntnis dar. Insbesonders da Python-*Threads* von sich aus keine wahre Gleichzeitigkeit unterstützen, liegt der Sinn in einer Verkleinerung der zu bearbeitenden Datenmenge, die dann dementsprechend früher abgefragt werden kann. Zur Untermauerung dieser These wurden auch die Laufzeiten der einzelnen *Threads* in obiger Analyse dargestellt. Der dabei erreichte Zeitgewinn von der Terminierung des ersten Threads bis hin zur Beendigung des gesamten sequentiellen Auslesevorgangs verhält sich aber unproportional zur Gesamtlaufzeit des gesamten *Thread*-Konstuktes. Die Anomalie, dass die Klasse Threaded_Transmitter mit zunehmendem Datenvolumen auf die Klasse Forked_Transmitter verliert, liegt in der Implementierung begründet. Durch den geteilten Adressraum müssten Threads eigentlich bei Performancevergleichen auf Einprozessorrechnern besser abschneiden. Die Klasse Threaded_Transmitter stützt sich jedoch auf ein *Thread-Framework* aus dem psycopg2 Datenbankadapter, das durch seine offene Programmierweise eine Vielzahl von Einsatzmöglichkeiten unterstützt, während Forked_Transmitter nur auf das Auslesen von x,y,z-Geometrien ausgelegt ist.
- Aus dem ersten Punkt ergibt sich, dass zwar das Einlesen parallel erfolgt, aber nicht durchgängig in der kompletten Visualisierungspipeline implementiert ist, wodurch sich erst ein Zeitgewinn ergeben würde. Momentan wird erst der Einlesevorgang bis zur korrekten Beendigung abgewartet, um dann weiter der Pipeline zu folgen.
- Notwendigerweise konzentrieren sich die Bemühungen auf die softwaretechnische Problematik ungeachtet der Hardware. Im Gesamtzusammenhang betrachtet, ergeben sich dabei jedoch praxisfremde Strukturen. An der Struktur von einer lokalen Datenbank sollte sich, um der Komplexität Herr zu werden, nichts ändern. Gängige Praxis sind aber *Cluster*-Strukturen aus mehreren Rechnern, die entsprechend ihrer Aufgabe CPU bzw. Speicher gemeinsam nutzen.
- Daraus ergibt sich weiter die unglückliche Trennung der einzelnen Aufgaben für die jeweiligen *Threads*. Sinn ist ja nicht nur eine Beschleunigung des Auslesevorgangs, sondern auch die Möglichkeit, Datenmengen, die die Größe des Hauptspeichers übersteigen, in Teilmengen zu prozessieren. Eine Trennung über die einzelnen Geometriespalten erzielt zwar bessere Ergebnisse als die Teilung über unterschiedliche räumliche Bereiche, da dabei die gesamte Datenbank abgefragt werden muss. Das gilt jedoch nur für Datenvolumina, die sich im Hauptspeicher verarbeiten lassen, da die einzelnen Ausleseergebnisse (das bedeutet in diesem Zusammenhang, dass sämtliche x,y,z Werte in lokalen *Container*-Objekten zwischengespeichert werden) wieder zusammengeführt werden müssen, was den Vorteil des „häppchen"-weisen Auslesens aufhebt. Aus heutiger Sicht würde sich ein System von mehreren Rechnern anbieten, das unterschiedliche Bereiche des zu visualisierenden Gebietes hält.

Einfache Designs können einfacher verstanden und optimiert werden. Einfachheit ist aber in Zusammenhang mit der komplexen Materie relativ zu beurteilen. Nichtsdestotrotz bleibt der Wert verteilter Rechenarchitekturen und damit verbundener paralleler Funktionalität unan-

getastet und der vielversprechendste Zugang, riesige Datenmengen zu verarbeiten. Nächster logischer Schritt wäre, eine geeignete Systemarchitektur zu finden und das Vorhandene dafür zu adaptieren.

Literatur

AVILA L.S., BARRÉ, S., BLUE, R., GEVECI, B., HENDERSON, W.A., KING, B., LAW, C.C., MARTIN, K.M. & W. SCHROEDER (2004): The VTK User's Guide. Updated for version 4.4. Kitware Inc., New York.

HÖFLE, B., GEIST, TH., HELLER, A. & J. STÖTTER (2005): Entwicklung eines Informationssystems für Laserscannerdaten mit OpenSource-Software. In: STROBL, J., BLASCHKE, T. & G. GRIESEBNER (Hrsg.): Angewandte Geoinformatik 2005. Beiträge zum 17. AGIT-Symposium Salzburg, 277-286. Wichmann Verlag, Heidelberg.

LUTZ, M. (2001): Programming Python. Solutions for Python Programmers. O'Reilly Media, Sebastopol.

MATLOFF, N. & F. HSU (2005): Introduction to Threads Programming with Python. URL: http://www.heather.cs.ucdavis.edu/~matloff/Python/PyThreads.pdf (08.09.2005).

SCHROEDER, W., L.S. AVILA & W. HOFFMAN (2005): The Visualization with VTK: A Tutorial. URL: http://www.caip.rutgers.edu/~silver/652/vtk_tutorial.pdf (17.09.2005).

SCHROEDER, W., MARTIN, K. & B. LORENSEN (2002): The Visualization Toolkit. An Object-Orientated Approach To 3D Graphics. o. A.

WEIGEND, M. (2005): Objektorientierte Programmierung mit Python. mitp Verlag, Bonn.