تعدادی از کتابخانه‌های ورودی/خروجی آسنکرون به برنامه نویس اجازه می‌دهند که درخواست یک تخمین از زمان کامل شدن یک عملیات آسنکرون خاص را بدهند. تعدادی از API ها به شما اجازه می‌دهند که ضرب الاجل را برای یک درخواست (که به صورت موثری درخواست را در مقایسه با سایر در خواست های منتظر اولویت می دهد) تعیین کنید و مشخص کنید که چه اتفاقی روی دهد هنگامی که یک درخواست ضرب الاجل را پشت سر بگذارد (مثلاً درخواست را لغو کند، به برنامه خبر دهد و به تلاشش ادامه دهد و غیره)

6.1.3.1

مهم است که به یاد داشته باشیم که ورودی/خروجی فایل یک سیستم زمان واقعی است (real time) است و مشابه بقیه ی بازی تحت تأثیر ضرب الاجل است. بنابراین عملیات های آسنکرون ورودی/خروجی معمولاً اولویت های متغیری دارند. برای نمونه اگر ما در حال جریان دادن (streaming) صوت از دیسک سخت یا بلوری باشیم و آن را در همان حال اجرا کنیم، بافر بعدی را پر از داده‌های صوتی می‌کند که مشخصاً اولویت بالاتری از مثلاً بارگزاری یک بافت یا یک بخش از مرحله ی بازی دارد. سیستم‌های ورودی/خروجی آسنکرون باید قادر به متوقف کردن درخواست های با اولویت پایین‌تر باشند تا درخواست های ورودی/خروجی با اولویت بالاتر شانسی برای تمام شدن پیش از ضرب الاجل داشته باشند.

6.1.3.2 چگونه ورودی/خروجی فایل آسنکرون کار می‌کند

ورودی/خروجی فایل آسنکرون با کنترل کردن درخواست های ورودی/خروجی در یک رشته ی جداگانه کار می کند. رشته ی اصلی تابعی را فراخوانی می‌کند که به سادگی درخواست هایی را بر روی صف قرار می‌دهد و سپس فوراً بازگردانی می کند. در همین حین رشته ی ورودی/خروجی درخواست هایی را از صف بر می‌دارد و به صورت دنباله ای آن‌ها را به استفاده از روتین های بلوکی ورودی/خروجی مانند read و write اجرا می‌کند. هنگامی که یک درخواست کامل شد یک فراخوانی به عقب متعلق به رشته ی اصلی فراخوانی می‌شود به این ترتیب به او اطلاع می‌دهد که عملیات تمام شده است. اگر رشته ی اصلی انتخاب کند که برای اتمام یک درخواست ورودی/خروجی صبر کند، این کار با یک سمافور انجام می شود. (هر درخواست یک سمافور مربوط به خود را دارد و رشته ی اصلی می‌تواند خود را در حالت خوابیده ی منتظر قرار دهد تا اینکه آن سمافور توسط رشته ی ورودی/خروجی بعد از تکمیل آن درخواست خبردار شود).

به صورت مجازی هر عملیات آسنکرون که شما می‌توانید تصور کنید می‌تواند به یک عملیات آسنکرون با انتقال کد به یک رشته ی جدا یا با اجرای آن بر روی یک پردازنده ی فیزیکی جداگانه مانند یکی از هسته ی های پردازنده مرکزی بر روی Play station 4، تبدیل شود. بخش ۷.۶ را برای جزییات بیشتر ببینید.

6.2 مدیر منابع

هر بازی از مجموعه وسیعی از منابع ساخته شده است (گاهی اوقات دارایی یا media خوانده می شود). مثالها شامل مش ها، مواد، بافتها، برنامه‌های سایه زنی، انیمیشن ها، کلیپهای صوتی، ساختار مرحله، برخوردهای اصلی، پارامترهای فیزیکی، و غیره است. یک منبع متعلق به بازی باید مدیریت شود، هم از نظر ابزارهای اداری برای ساخت آن‌ها و هم از نظر بارگزاری، بارگزاری به عقب و کنترل آن‌ها در زمان اجرا. بنابراین هر موتور بازی یک مدیر منابع از یک نوع دارد.

هر مدیر منبع شامل دو بخش جدا ولی به هم مرتبط است. یک بخش زنجیره ی ابزارهای آفلاین مورد استفاده برای ساخت دارایی ها و تبدیل آن‌ها به فرم آماده برای موتور را مدیریت می کند. بخش دیگر منابع را در زمان اجرا مدیریت می‌کند و اطمینان حاصل می‌کند که به ترتیبی که توسط بازی مورد نیاز هستند در حافظه بارگزاری شده اند. و همچنین مطمین می‌شود که هنگامی که مورد نیاز نیستند از حافظه خارج می شوند.

در تعدادی از موتورها، مدیر منبع یک زیرسیستم به خوبی طراحی شده، یکی شده و مرکزی شده است که تمام انواع منابع مورد استفاده توسط بازی را مدیریت می کند. در سایر موتورها مدیر منبع به عنوان یک زیرسیستم یکتا وجود ندارد بلکه بین مجموعه‌ای اززیرسیستم ها تقسیم شده است که احتمالاً توسط افراد مختلفی در زمانهای مختلفی در طول تاریخچه ی طولانی و بعضی اوقات رنگارنگ موتور نوشته شده است. اما مهم نیست که چگونه پیاده‌سازی شده است، یک مدیر منبع به صورت بدون تغییری تعدادی مسئولیت دارد و تعدادی از مسایل به خوبی فهمیده شده را حل می کند. در این بخش ما کارایی و تعدادی از جزییات پیاده‌سازی یک مدیر منابع موتور بازی معمولی را توضیح می دهیم.

6.2.1 مدیریت منابع آفلاین و زنجیره ابزار

6.2.1.1 کنترل اصلاح برای دارایی ها

در یک پروژه بازی کوچک، دارایی های بازی می‌تواند مدیریت شود با حفظ فایل‌های که در یک درایو اشتراک یافته ی شبکه با یک ساختار دایرکتوری موردی وجود دارند. این روش برای بازی‌های سه بعدی تجاری مدرن عملی نیست زیرا شامل تعداد زیاد و متفاوتی از دارایی ها هستند. برای چنان پروژه ای، تیم نیازمند یک روش بیشتر فرمال شده برای تعقیب و مدیریت دارایی ها است.

تعدادی از تیم های بازی سازی از یک سیستم کنترل بازبینی برای مدیریت منابعشان استفاده می کنند. فایل منبع های هنری (صحنه های مایا، فایل‌های فوتوشاپ، فایل‌های ایلوستریتور و غیره…) توسط هنرمندان به درون پرفورس یا یک بسته ی مشابه وارد می شوند. این روش به صورت معقولی خوب کار می کند. هرچند تعدادی از تیم های بازی سازی ابزارهای سفارشی ی مدیریت دارایی دارند که به آن‌ها کمک می‌کند تا منحنی یادگیری را برای هنرمندانشان صافتر کنند. چنان ابزارهایی می‌تواند به راحتی در اطراف یک سیستم کنترل بازبینی قرار بگیرد یا اینکه سیستم تماماً سفارشی باشد.

برخورد با اندازه ی داده

یکی از بزرگترین مشکلات در کنترل بازبینی برای دارایی های هنری اندازه ی زیاد داده است. در حالی که کد منبع های cpp و اسکریپت ها در مقایسه با تاثیرشان بر روی پروژه کوچک هستند، فایل‌های هنری تمایل دارند که بسیار بسیار بزرگ‌تر باشند. به دلیل اینکه بسیاری از سیستم‌های کننرل منبع با کپی کردن فایل‌ها از ریپازیتوری مرکزی به ماشین محلی کاربر کار می کنند، اندازه ی بزرگ فایل‌های دارایی ها می‌تواند این بسته ها را کاملاً بلا استفاده کند.

من تا کنون تعداد از راه حل‌های متفاوت را برای این حل این مشکل در استودیوهای مختلف دیده ام. تعدادی از استودیو ها به سیستم‌های کنترل بازبینی تجاری مانند ایلین برین که به صورت اختصاصی برای کارکردن با اندازه های بزرگ داده طراحی شده‌اند،‌روی آورده اند. تعدادی از تیم ها به سادگی ٬بسته های حجیمشان را می گیرند٬ و به ابزارهای کنترل بازبینی شان اجازه می‌دهند تا دارایی ها را به صورت محلی کپی کند. این کار تا زمانی مؤثر است که دیسک های شما به اندازه کافی بزرگ است و پهنای باند شبکه شما کافی است اما در عین حال می‌تواند غیرکارا باشد و سرعت تیم را پایین آورد. تعدادی از تیم ها از سیستم‌های پیچیده‌ای بر روی ابزارهای کنترل بازبینی استفاده می‌کنند برای اطمینان از اینکه تنها کاربر کپی های محلی از فایل‌هایی به دست می‌آورد که واقعاً مورد نیاز هستند. در این مدل یا کاربر هیچ دسترسی به بقیه ی ریپازیتوری ندارد و یا می‌تواند در صورت نیاز به آن‌ها در یک درایو شبکه ی مشترک دسترسی داشته باشد.

در استودیوی Naughty dog ما از یک ابزار مناسب استفاده می‌کنیم که از لینک های سمبلیک یونیکس برای حذف مجازی کپی داده استفاده می‌کند در حالی که به هر کاربر اجازه می‌دهد که یک دید محلی کامل از ریپازیتوری دارایی ها داشته باشد. تا زمانی که یک فایل برای ویرایش انتخاب نشود تنها یک لینک سمبلیک به یک فایل اصلی در یک درایو شبکه ی مشترک است. یک لینک سمبلیک فضای بسیار کمی را بر روی دیسک محلی می‌گیرد زیرا چیزی بیش از یک ورودی دایرکتوری نیست. هنگامی که کاربر یک فایل را بری ویرایش انتخاب می‌کند، لینک سمبلیک حذف می‌شود و یک کپی محلی از فایل جایگزین آن می شود. هنگامی که کاربر ویرایشش را تمام کرد و فایل را در جای خود بازگرداند، کپی محلی تبدیل به کپی اصلی جدید می شود، تاریخچه ی بازبینی اش در یک پایگاه داده ی مرکزی به روز رسانی می‌شود و فایل محلی دوباره تبدیل به یک لینک سمبلیک می شود. این سیستم بسیار خوب کار می‌کند اما تیم را مجبور می‌کند تا سیستم کنترل بازبینی شان را از کف بنویسند؛ من هیچ ابزار تجاری را نمی‌شناسم که این‌گونه کار کند. همچنین لینک های سمبلیک یکی از امکانات یونیکس هستند-- چنان ابزاری را به احتمال زیاد می‌شود با junction های ویندوز (معادل لینک سمبلیک ویندوز) ساخت اما تا کنون ندیده‌ام که کسی برای انجام این کار تلاش کرده باشد.

6.2.1.2 پایگاه داده ی منابع

همان‌گونه که در بخش آینده به صورت عمیق در مورد آن بحث می کنیم، بیشتر دارایی ها به همان صورت اولیه شان توسط موتور بازی مورد استفاده قرار نمی گیرند. آن‌ها نیازمند این هستند که از تعدادی از خط لوله های تغییر حالت دارایی، که کارشان تبدیل دارایی به فرمت دودویی مورد نیاز توسط موتور است، عبور کنند. برای هر منبعی که از این خط لوله عبور می کند، تعدادی متاداده که چگونگی پردازش منبع را توضیح می‌دهد موجود است. هنگامی که یک نقشه می بیتی (بیت مپ) را فشرده‌سازی می کنیم، ما نیاز داریم که بدانیم چه نوعی از فشرده‌سازی برای آن عکس خاص مناسب‌تر است. هنگام خروجی گرفتن از یک انیمیشن، ما نیاز داریم که بدانیم چه دامنه ای از فریم ها در مایا باید خروجی گرفته شوند. هنگام خروجی گرفتن از مش های کاراکترها از یک صحنه ی مایا که چندین کاراکتر را در خود دارد، باید بدانیم که کدام مش مطابق کدام کاراکتر در بازی است.

برای مدیریت تمام این متاداده ها ما نیازمند توعی از پایگاه داده هستیم. اگر ما یک بازی بسیار کوچک می سازیم، این پایگاه داه ممکن است در مغز خود توسعه دهندگان باشد. من صدایشان را اکنون می‌شنوم که می گویند: یادت باشد: انیمیشن های بازیکن باید فلگ چرخاندن x اش تنظیم شده باشد، اما برای کاراکتر دیگر نباید تنظیم شده باشد یا … موشها … آیا راه دیگری است؟

به وضوح برای هر بازی با اندازه ی قابل قبول ما به راحتی نمی‌توانیم بر روی حافظه های طراحان مان به این صورت تکیه کنیم. به این دلیل که اندازه ی عظیم دارایی ها تقریباً به سرغت بسیار زیاد می شود. پردازش کردن قایل های منبع فردی با دست همچنین بسیار زمان برتر است که در تولید یک بازی تجاری با مقیاس کامل قابل انچام باشد. بنابراین هر تیم بازی سازی تخصصی تعدادی از خط لوله های منابع نیمه اتوماتیک دارد و داده‌ای که این خط لوله را می راند در نوعی از پایگاه داده ی منبع ذخیره شده است.

این پایگاه داده ی منبع فرم‌های بسیار متفاوتی را در موتور های بازی سازی متفاوت می گیرد. در یک موتور، متاداده ای که چگونگی ساخت یک منبع را توضیح می‌دهد ممکن است به صورت توکار درون خود دارایی های منبع باشد (مثلاً ممکن است به عنوان داده ی کور درون یک فایل مایا باشد). در موتور دیگری هر فایل منبع ممکن است با یک فایل متنی همراه شود که چگونگی پردازش آن را توضیح می دهد. با وجود این سایر موتور ها متاداده ی ساخت منبعشان را در یک مجموعه از فایل‌های xml، که احتمالاً درون نوع از اینترفیس گرافیکی کاربر پیچیده شده‌اند کد می کنند. تعدادی از موتورها از یک پایگاه داده ی رابطه‌ای واقعی استفاده می‌کنند مانند Microsoft access و MySql یا حتی یک پایگاه داده ی سنگین وزن مانند Oracle.

هرچه که فرم آن باشد، یک پایگاه داده ی منبع باید این کارایی های پایه را ارایه دهد:

* توانایی برخورد با انواع مختلفی از منابع، به صورت ایده‌آل (اما قطعاً نه به صورت الزامی) به شکلی تقریباً غیرمتناقض
* توانایی ساخت منابع جدید
* توانایی حذف منابع
* توانایی تشخیص و ویرایش منابع موجود
* توانایی انتقال یک فایل منبع از یک محل به محل دیگر بر روی دیسک (این بسیار یاری رسان است زیرا هنرمندان و طراحان بازی معمولاً نیازمند تغییر چینش دارایی ها برای انعکاس اهداف متغیر پروژه، بازفکری طراحی های بازی، اضافه کردن امکانات و حذف … هستند)
* توانایی یک منبع برای ارجاع کردن یه سایر منابع (مثلاً مواد استفاده شده توسط یک مش یا مجموعه ی انیمیشن های مورد نیاز توسط مرحله ی ۱۷). این اشاره کردن بخصوص هردو فرایند ساخت منبع و بارگزاری را در زمان اجرا کنترل می کند.
* توانایی نگهداری یکپارچگی مرجعی تمام ارجاع دهی ها درون پایگاه داده و یا برای انجام آن در ظاهر تمام عملیات های مشترک مانند حذف یا انتقال منابع به اطراف.
* توانایی نگهداری یک تاریخچه ی بازبینی کامل همراه با اطلاعاتی از جمله اینکه چه کسی هر کدام از تغییرات را انجام داده است و چرا
* همچنین بسیار یاری رسان است اگر پایگاه داده ی منابع جستجو یا پرس و جو (query) را به روش‌های گوناگون پشتیبانی کند. برای نمونه یک توسع دهنده ممکن است نیازمند دانستن این باشد که در کدام مرحله یک انیمیشن خاص مورد استفاده قرار گرفته است یا کدام بافت توسط مجموعه‌ای از مواد مورد ارجاع قرار گرفته است. یا اینکه آن‌ها ممکن است سعی در یافتن یک منبع بکنند که گاهی اوقات اسمش را فراموش می کنند.

باید کاملاً واضح باشد که از نگاه کردن به لیست بالا فهمیده می‌شود که ساختن یک پایگاه داده ی منابع قابل اعتماد و سریع یک وظیفه ی آسان نیست. هنگامی که به خوبی طراحی شود و به درستی پیاده‌سازی شود، پایگاه داده ی منابع می‌تواند کاملاً تفاوت بین یک تیم را که یک بازی عالی را ایجاد می‌کند با یک تیم که خود را ۱۸ ماه مشغول می‌کند و در نهایت توسط مدیران مجبور به توقف پروژه می‌شود (یا بدتر) ایجاد کند. من می‌دانم که این درست است زیرا من شخصاً هر دو حالت را تجربه کرده ام.

6.2.1.3 تعدادی از طراحی های موفق پایگاه داده ی منابع

هر تیم بازی سازی نیازمندی های متفاوتی دارد و تصمیمات متفاوتی را در هنگام طراحی پایگاه داده ی منابعشان اتخاذ می کنند. هرچند برای آن مواردی که می ارزد، تعدادی از طرح هایی که به تجربه ی من خوب کار کرده‌اند آمده است.

Unreal Engine 4

پایگاه داده ی منابع آنریل توسط uber-tool و UnrealEd مدیریت می شود. UnrealEd مسئول تقریباً همه چیز است، از مدیریت متاداده ی منابع تا ساخت دارایی تا طراحی ظاهر مرحله و بیشتر. UnrealEd دارای مشکلات مربوط به خود نیز هست، اما یکی از بزرگترین نقاط قوت آن این است که خود یک بخش از موتور بازی است. این به دارایی ها اجازه می‌دهد که ایجاد شوند و سپس فوراً در نهایت شکوهشان به نمایش درآیند، دقیقاً به همان صورتی که درون بازی به نمایش در می آیند. بازی حتی می‌تواند از درون UnrealEd اجرا شود تا دارایی ها را در محیط طبیعیشان به نمایش بگذارد و ببینیم که چگونه درون بازی نشان داده می شوند.

یکی دیگر از فواید بزرگ UnrealEd چیزی است که من خرید یک مرحله‌ای می نامم. براوزر اصلی UnrealEd (که در شکل ۶.۱ نشان داده شده است) به توسعه دهندگان اجازه می‌دهد که تقزیبا به تمامی منابعی که توسط موتور مصرف شده‌اند دسترسی داشته باشند. داشتن یک اینترفیس پایدار، یکتا و یکی شده برای ساخت و مدیریت تمام انواع منابع یک موفقیت بزرگ است. این مخصوصاً با در نظر گرفتن اینکه داده‌های منابع در بیشتر موتورهای بازی در میان ابزارهای بیشمار نامنطبق و معمولاً رمز شده پخش شده است به چشم می آید. تنها قادر بودن به اینکه هر منبعی را به آسانی در UnrealEd پیدا کرد خود یک مثبت بزرگ است.

آنریل می‌تواند نسبت به بسیاری از موتورهای دیگر کمتر مشکل خیز باشد، زیرا دارایی ها باید به صورت صریح به درون پایگاه داده ی منابع آنریل وارد شوند. این به منابع اچازه می‌دهد تا در همان ابتدای فرایند ساخت برای معتبر بودن مورد آزمایش قرار گیرند. در بیشتر موتورهای بازی، هر داده ی قدیمی می‌تواند به درون پایگاه داده ی منابع انداخته شود و شما تنها هنگام ساخت نهایی آن داده می‌دانید که معتبر بوده است – یا بعضی اوقات تا زمان بارگزاری نهایی در هنگام اجرای بازی متوجه نمی شوید. اما با آنریل، دارایی ها می‌توانند در همان حین ورود به UnrealEd اعتبارسنجی شوند. این بدین معناست که فردی که دارایی را ساخته است بازخورد فوری را در مورد اینکه آیا دارایی اش به درستی تنظیم شده است یا نه دریافت می کند.

البته روش آنریل مشکلات جدی ای هم دارد. برای نمونه، تمام داده‌های منابع در تعداد کوچکی از فایل‌های بسته ی بزرگ ذخیره می شود. این فایلها دودویی هستند، بنابراین به آسانی با یک بسته ی کنترل بازبینی مانند cvs، Subversion یا Perforce ادغام نمی شوند.این تعدادی مشکلات بزرگ را در هنگامی که بیشتر از یک کاربر می‌خواهد منبعی را ویرایش کند که در یک بسته ی یکتاست، ایجاد می کند. حتی اگر کاربران سعی در ویرایش منابع مختلفی داشته باشند، تنها یک کاربر می‌تواند در یک زمان بسته را قفل کند بنابراین دیگران باید منتظر بمانند. شدت این مشکل می‌تواند با تقسیم منابع به بسته های تقریباً کوچک و دانه بندی شده کمتر شود اما به صورت عملی نمی‌تواند حذف شود.

تمامیت ارجاعی در UnrealEd تقریباً خوب است اما هنوز هم مشکلاتی وجود دارند. هنگامی که یک منبع تغییر نام داهد می‌شود و یا به جای دیگری منتقل می شود، تمام ارجاع های به آن به صورت اتوماتیک با استفاده از یک شی ساختگی که منابع قدیمی را به مکان/اسم جدید آن تغییر آدرس می دهد، تغییر می کند. مشکل با این اشیاء ساختگی تغییر آدرس این است که آن‌ها تبدیل به باری بر روی سیستم می‌شوند و جمع می‌شوند و گاهی اوقات مشکلاتی را ایجاد می کنند، بخصوص اگر یک منبع حذف شود. در کل، تمامیت ارجاعی آنریل تقریباً خوب است اما عالی نیست.

با وجود مشکلاتش، UnrealEd تاکنون برای کاربران مناسبترین مجموعه ابزار ساخت دارایی، پایگاه داده ی منابع و خط لوله ی تبدیل دارایی ای بوده است که با آن کار کرده ام، همچنین بسیار مجتمع و منظم هم هست.

موتور مربوط به بازی‌های Uncharted و The last of us متعلق به استودیوی Naughty dog

برای Uncharted: Drakes Fortune، ناتی داگ متاداده های منابعش را در یک پایگاه داده ی MySQL ذخیره کرد. یک ظاهر کاربری گرافیکی سفارشی هم برای مدیریت محتوای پایگاه داده نوشته شد. این ابزار به هنرمندان، طراحان بازی و برنامه نویسان به صورت مشابهی امکان می‌داد تا منابع جدید بسازند، آن‌ها را حذف کنند و همچنین منابع را جستجو و ویرایش کنند. این GUI یک بخش حیاتی سیستم بود، زیرا به کاربران امکان می‌داد تا دیگر مجبور به آموزش پیچیدگی های تعامل با یک پایگاه داده ی رابطه‌ای مانند SQL را نداشته باشند.

پایگاه داده ی MySql اولیه که در Uncharted مورد استفاده قرار گرفت، یک تاریخچه ی کامل از تغییرات انجام شده بر روی پایگاه داده را تأمین نمی‌کرد و همچنین یک راه خوب برای بازگرداندن تغییرات هم نداشت. همچنین چند کاربر که یک منبع را ویرایش کنند را پشتیبانی نمی‌کرد و برای مدیر هم دشوار بود. ناتی داگ از آن زمان به نفع پایگاه داده‌های دارایی بر اساس فایل XML که تحت Perforce مدیریت می شوند، از MYSql دست کشید.

‌Builder، ظاهر گرافیکی کاربری پایگاه داده ی منابع ناتی داگ در شکل ۶.۲ به نمایش درآمده است. پنجره به دو بخش اصلی تقسیم شده است: یک درخت که تمام منابع را در بازی در سمت چپ نمایش می‌دهد و یک پنجره ی مشخصات در سمت راست که به منابعی که در پنجره ی درخت انتخاب شده‌اند اجازه می‌دهد تا دیده و ویرایش شوند. درخت منبع شامل پوشه هایی برای اهداف سازمانی است، تا هنرمندان و طراحان بازی بتوانند منابعشان را به هر صورتی که مناسب می‌بینند سازمان دهی کنند. انواع مختلفی از منابع می‌تواند درون هر پوشه ساخته و ویرایش شوند شامل عملگرها و مراحل و زیرروتین هایی که درون آن‌ها هستند (معمولاً مش ها، اسکلت ها و انیمیشن ها). انیمیشن ها همچنین می‌توانند به درون پوشه مانندهایی تحت عنوان مجموعه گروه بندی شوند. این به گروه‌های بزرگ از انیمیشن ها اجازه می‌دهد که ساخته شوند و سپس به عنوان یک واحد مدیریت شوند و از مقدار زیادی هدر رفت وقت به دلیل جابه جایی تک انیمیشن ها به اطراف جلوگیری می کند.

خط لوله ی تبدیل دارایی که در آنچارتد و The last of us مورد استفاده قرار گرفت، شامل مجموعه‌ای ازخارج کننده های منابع، کامپایلر و لینکر است که از محیط دستوری فراخوانی می شوند. موتور توانایی کار با مجموعه ی زیادی از انواع مختلف اشیاء داده‌ای را دارد اما این‌ها به درون یکی از دو نوع فایل‌های منبع بسته بندی شده اند: عملگر و مرحله ها. یک عملگر می‌تواند شامل اسکلت، مش، مواد، بافت و/یا انیمیشن باشد. یک مرحله شامل مش های ثابت پیش‌زمینه، مواد و بافت است و همچنین اطلاعات مربوط به طرح اطلاعاتی مرحله را هم در خود دارد. برای ساخت یک عملگر، به سادگی می‌توان یک ba نام عملگر را بر روی محیط دستوری تایپ کرد؛ برای ساخت یک مرحله باید bl نام مرحله را تایپ می کنیم. این ابزارهای محیط متنی، پایگاه داده را برای تعیین دقیق چگونگی ساخت یک عملگر یا مرحله مورد درخواست، جستجو می کنند. این شامل اطلاعات در مورد چگونگی استخراج دارایی از ابزاهای DCC مانند مایا و فوتوشاپ، چگونگی پردازش داده ها، و چگونگی بسته بندی آن به صورت فایل دودویی .pak است که می‌توانند توسط موتور بازی بارگزاری شوند. این بسیار ساده‌تر از بسیاری از موتورها که منابع باید به صورت دستی توسط هنرمند استخراج شوند-- یک وظیفه ی وقت گیر، خسته‌کننده و مستعد خطا.

منافع طرح خط لوله ی منابع استفاده شده توسط ناتی داگ شامل:

* منابع دانه بندی شده: منابع می‌توانند به عنوان موجودیت های منطقی در بازی -مش ها، مواد، اسکلت ها و انیمیشن ها- مورد استفاده قرار گیرند. این نوع منابع به اندازه کافی اندازه بزرگی دارند که تیم تقریباً هیچ گاه هیچ اختلاف نظری در مورد اینکه کدام دو کاربر باید به صورت هم‌زمان آن را ویرایش کنند،‌پیدا نمی کند.
* امکانات ضروری (و هیچ چیز دیگر). ابزار سازنده یک مجموعه ی قدرتمند از امکانات را به ما می‌دهد که نیازهای تیم را برطرف می کند، اما ناتی داگ هیچ منبع ی را برای ساخت امکاناتی که مورد نیازشان نباشد هدر ندادند.
* نگاشت واضح به یک فایل منبع. یک کاربر می‌تواند بسیار سریع تشخیص دهد که کدام دارایی منبع (فایل های محلی DCC، مانند مایا یا فوتوشاپ) یک منبع را تشکیل می دهند.
* تغییر روش استخراج و وارد کردن داده‌های DCC ساده است. تنها با کلیک بر روی منابع مورد نظر و تغییر مشخصات پردازشی درون ظاهر کاربری پایگاه داده ی منبع
* ساخت دارایی ساده است. تنها با تایپ ba یا bl که با نام منبع ادامه پیدا کند، بر روی یک محیط متنی. سیستم وابستگی بقیه ی کارها را انجام می‌دهد

البته زنجیره ابزار ناتی داگ همچنین مشکلاتی دارد شامل:

* عدم وجود ابزار نمایش. تنها راه برای نشان دادن یک دارایی این است که آن را به درون بازی یا نمایش دهنده ی مدل/انیمیشن بارگزاری کرد (که تنها یک حالت خاص از خود بازی است)
* ابزار کاملاً مجتمع نشده است. ناتی داگ از یک ابزار برای نمایش مراحل استفاده می کند، یکی دیگر برای مدیریت اکثریت منابع در پایگاه داده ی منابع، و یک مورد سوم برای آماده کردن مواد و سایه زن‌ها (این یک بخش از قسمت جلویی پایگاه داده ی منابع است). ساخت دارایی در محیط متنی انجام می شود. ممکن است که راحت‌تر باشد اگر تمام این کارایی ها به درون یک ابزار واحد مجتمع می شدند. هرچند ناتی داگ هیچ برنامه‌ای برای انجام این ندارد، زیرا فواید به احتمال زیاد هزینه‌های ایجاد شده را جبران نمی کند.

سیستم مدیریت منابع OGRE

OGRE یک موتور رندر کردن است، نه یک موتور بازی کامل. به این معنی که اوگر دارای یک مدیر منابع در حین اجرای کامل به خوبی طراحی شده و کامل نیست. یک اینترفیس پایدار و ساده برای بارگزاری تقریباً هر نوع از منابع مورد استفاده قرار می گیرد. و سیستم با در نظر گرفتن توسعه پذیری طراحی شده است. هر برنامه نویس می‌تواند تقریباً به آسانی بک مدیر منابع را برای یک نوع جدید از دارایی‌ها پیاده‌سازی کند و به آسانی آن را به درون چارچوب منابع اوگر وارد کند.

یکی از مشکلات مدیر منابع اوگر این است که تنها در هنگام اجرا کار می کند. اوگر فاقد هر نوع پایگاه داده ی منابع آفلاین است. اوگر تعدادی استخراج کننده که قادر به تبدیل یک فایل مایا به یک مش که قابل استفاده توسط اوگر(کامل با مواد، سایه زنها، یک اسکلت و انیمیشن های اختیاری) باشد را ندارد. هرچند استخراج کننده باید به صورت دستی از درون خود مایا به کار انداخته شود. بدتر از همه متاداده هایی که چگونگی استحراخ و پردازش یک فایل مایا را توضیح می‌دهند باید توسط کاربری که عمل استخراج را انجام می‌دهد وارد شوند.

به صورت خلاصه، مدیر منابع در زمان اجرای اوگر، قدرتمند و به خوبی طراحی شده است. اما اوگر به همان اندازه از یک پایگاه داده منابع و خط لوله ی تغییر دارایی مدرن و قدرتمند در قسمت ابزارها سود خواهد برد.

XNA متعلق به مایکروسافت

XNA یک مجموعه ابزار توسعه بازی متعلق به مایکروسافت است که هدفش را بر روی پلتفرم های PC و XBOX ۳۶۰ قرار داده است. سیستم مدیریت منابع XNA یکتا است، به این دلیل که مدیریت پروژه و ساخت سیستم‌ها را برای مدیریت و همچنین ساخت دارایی به Visual Studio IDE واگذار می کند. ابزار توسعه ی بازی XNA به نام Game Studio Express تنها یک افزونه برای Visual Studio Express است. شما می‌توانید بیشتر در مورد Game Studio Express در سایت مایکروسافت مطالعه کنید.

6.2.1.4 خط لوله ی تبدیل دارایی

در بخش 1.7 ما یاد گرفتیم که داده‌های منابع معمولاً با استفاده از ابزارهای پیشرفته ی ساخت محتوای دیجیتالی (DCC) مانند مایا، Zbrush، فوتوشاپ یا Houdini‌ساخته می شوند. هرچند فورمت داده‌ای که توسط این ابزارهای مورد استفاده قرار می‌گیرد معمولاً برای مصرف مستقیم توسط موتور بازی مناسب نیست. بنابراین اکثریت داده‌های منابع از یک خط لوله ی تبدیل دارایی (ACP) در مسیرشان به موتور بازی عبور می کنند. خط لوله گاهی اوقات به عنوان خط لوله ی دارایی شناخته می‌شود (RCP) یا خیلی ساده زنچیره ابزار.

هر خط لوله ی منابع با یک مجموعه از دارایی‌ها در شکل اولبه ی DCC (مثلاً مایا .ma یا .mb، فایل‌های .psd فوتوشاپ)هستند. این دارایی‌ها معمولاً از سه مرحله ی پردازش در مسیرشان به طرف موتور بازی عبور می کنند:

1. استخراج کننده. ما نیازمند راه‌هایی برای گرفتن داده از فورمت اولیه ی DCC و تبدیل آن به یک فورمتی که موی توانیم مورد استفاده قرار دهیم، هستیم. این معمولاً با نوشتن یک افزونه ی سفارشی برای DCC مورد نظر انجام می شود. این کار افزونه است که داده را به یک نوع فایل میانی تبدیل کند که بتوان در مراحل بعدی خط لوله فرستاده شود. بیشتر کاربردهای DCC یک مکانیزم به اندازه کافی آسان را برای انجام این ارایه می دهند.

مایا در حقیقت سه چیز را ارایه می دهد: یک SDK متعلق به cpp، یک زبان اسکریپتی که به عنوان MEL شناخته می‌شود و اخیراً هم یک اینترفیس پایتون.

در مواردی که یک برنامه ی DCC هیچ قلاب شخصی سازی به ما نمی دهد، ما می‌توانیم همواره داده‌ها را به عنوان یکی از فورمت های محلی ابزار DCC ذخیره کنیم.در هر صورت یکی از این‌ها یک فورمت باز خواهد بود، یک فورمت متنی به اندازه کافی بصری یا فورمت های دیگری که می‌تواند مورد مهندسی معکوس قرار گیرد. با در نظر گرفتن این مورد، ما می‌توانیم فایل را به صورت مستقیم به مرحله ی بعدی از خط لوله بفرستیم.

1. کامپایلرهای منابع. ما معمولاً باید داده‌های خام استخراج شده از یک برنامه ی DCC را به روش‌های گوناگون ورز بدهیم تا برای بازی آماده شوند. برای نمونه ما ممکن است نیاز به بازچینی یک مش مثلث به بخش‌های آن، یا فشرده‌سازی یک بیت مپ بافت یا محاسبه ی اندازه ی منحنی بخشی از اسپلاین Catmull-Rom داشته باشیم. تمام منابع نیاز به کامپایل شدن ندارند- بعضی‌ها ممکن است به محض استخراج برای بازی آماده باشند.
2. اتصال دهنده های منابع. چندین فایل منبع گاهی اوقات نیاز به ترکیب شدن به یک بسته ی سودمند یکتا،‌پیش از بارگزاری شدن توسط موتور بازی دارند. این فرایند اتصال فایل‌های آبجکت به هم در هنگام کامپایل یک برنامه ی Cpp را به یک فایل قابل اجرا تقلید می‌کند و بنابراین این فرایند گاهی اوقات اتصال منابع نام دارد. برای نمونه هنگام ساخت یک منبع مرکب پیچیده مانند یک مدل سه بعدی ما ممکن است نیاز به ترکیب داده‌ها از چندین فایل مش استخراج شده، چندین فایل مواد، یک فایل اسکلت و چندین فایل انیمشن به یک منبع یکتا، داشته باشیم. تمام انواع منابع نیاز به اتصال ندارند-- بعضی دارایی‌ها بعد از مراحل استخراج یا کامپایل آماده برای بازی هستند.

وابستگی‌های منابع و قواعد ساخت

بسیار شبیه کامپایل فایل منبع در یک پروژه ی c یا cpp و سپس اتصال آن‌ها برای ساخت یک فایل قابل اجرا، خط لوله ی تبدیل دارایی، دارایی‌های منبع را (به شکل فایل‌های هندسه یا انیمیشن مایا، فایل‌های فوتوشاپ، کلیپ های خام صوتی، فایل‌های متنی و …) پردازش می‌کند و آن‌ها را به شکل آماده برای بازی در می‌آورد و سپس به یک شکل همجنس برای استفاده توسط موتور در می آورد. و دقیقاً مانند فایل‌های منبع در یک برنامه ی کامپیوتری، دارایی‌های بازی معمولاً باهم وابستگی دارند. (برای مثال یک مش به یک یا چند ماده اشاره می‌کند که خود به چند بافت اشاره می کند). این وابستگی‌های درونی معمولاً یک تأثیر را بر ترتیبی که دارایی‌ها باید توسط خط لوله پردازش شوند می گذارد. (برای مثال، ما ممکن است نیاز به ساخت یک اسکلت کاراکتر پیش از پردازش هر کدام از انیمیشن های آن کاراکتر داریم) به علاوه، وابستگی‌های بین دارایی‌ها به ما می‌گوید که کدام دارایی باید هنگامی که یک دارایی منبع خاص تغییر کرد، دوباره سازی شود.

وابسنگی های ساخت نه تنها بر تغییر به خود دارایی‌ها تأثیر دارند بلکه همچنین به تغییر به فورمت های داده‌ای هم وابسته هستند. اگر برای مثال فورمت تمام فایل‌های استفاده شده برای ذخیره ی مش های مثلث تغییر کند، تمام مش ها در تمام بازی ممکن است نیاز به دوباره استخراج و دوباره سازی داشته باشند. بعضی موتورهای بازی فورمت های داده‌ای را مورد استفاده قرار می‌دهند که در برابر تغییرات نسخه مقاوم هستند. برای مثال یک دارایی ممکن است شامل یک شماره ی نسخه باشد و موتور بازی ممکن است شامل کدی باشد که می‌داند چگونه از دارایی‌های قدیمی استفاده کند و آن‌ها را بارگزاری کند. مشکلات چنان سیاستی این است که فایل دارایی و کد موتور تمایل دارند که بسیار حجیم شوند. هنگامی که تغییر فورمت های داده‌ای بسیار نادر است، ممکن است بهتر باشد که تنها گلوله را گاز گرفت و تمام فایل‌ها را هنگامی که تغییر فورمت روی می‌دهد، دوباره پردازش کرد.

خط لوله های ساده ی دارایی نیاز به مجموعه‌ای از قوانین دارند که وابستگی‌های درونی را بین دارایی‌ها توضیح می‌دهد و نوعی از ابزار ساخت که می‌تواند از این اطلاععات برای اطمینان از اینکه یک دارایی خاص، به ترتیب مناسب ساخته شده، در هنگام ویرایش یک دارایی منبع استفاده کند. تعدادی تیم های بازی سیستم ساخت خودشان را می سازند. سایرین از یک ابزار تثبیت شده استفاده می‌کنند مانند make. هر راه حلی که انتخاتب شود،‌تیم باید با سیستم وابستگی ساختشان با احتیاط کامل رفتار کنند. اگر شما اینگونه نکنید، تغییرات به کد منبع دارایی ممکن است باعث نشود که دارایی مناسب دوباره سازی شود. نتیجه می‌تواند با دارایی‌های بازی متناقض باشد، که خود ممکن است باعث موارد غیر طبیعی بصری شود و یا حتی باعث کرش موتور شود. به تجربه ی شخصی، من ساعات بیشماری را دیده‌ام که صرف پیدا کردن مشکلاتی شده است که می‌توانست پیشگیری شود اگر وابستگی‌های درونی دارایی‌ها به خوبی مشخص می‌شد و سیستم ساخت برای استفاده ی آن‌ها به صورت قابل اعتمادی پیاده‌سازی می شد.

6.2.2 مدیریت منابع در هنگام اجرا

اجازه دهید که توجهمان را به معطوف کنیم که چگونه دارایی‌ها در پایگاه داده ی منابع ما بارگزاری می شوند، مدیریت می‌شوند و از موتور خارج می‌شوند در هنگام اجرا.

6.2.2.1

یک مدیریت منابع موتور بازی در هنگام اجرا مجموعه ی وسیعی از وظایف را بر عهده دارد، که همه آن‌ها مرتبط با وظیفه ی اصلی آن یعنی بارگزاری منابع به درون حافظه است:

* اطمینان از اینکه تنها یک کپی از هر منبع واحد در هر زمان در حافظه موجود است.
* مدیریت زمان حیات هر منبع
* بارگزاری منابع مورد نیاز و خارج کردن منابعی که دیگر مورد نیاز نیستند.
* کنترل بارگزاری منابع مرکب. یک منبع مرکب یک منبع است که شامل سایر منابع است. برای مثال یک مدل سه بعدی یک منبع مرکب است که شامل یک مش، یک یا بیشتر ماده، یک یا بیشتر بافت و به صورت دلخواه یک اسکلت و چندین انیمیشن اسکلت است.