This is the Cover Letter for my application for a games industry tools programmer role.

I want to be a programmer in the games industry because the games industry is my passion ever since the Sinclair ZX Spectrum days. The good old days of Dizzy series, Elite and Knight Lore and Z80 Basic's UDG :)

I want to be a tools programmer because, as a generalist programmer, a tools – related role seems like the easy way into any company. I did work for tools when at Frontier Developments for their then new(er) Outsider, which did not get released but sure enough prepared the back-end for the newest Elite. It was a standalone program that allowed an artist to edit the attributes of some assets from the game, all complete with game paths and general integration with the actual asset pipeline. The assets themselves were having the in-game instance attributes stored in some xml type of files with the .oct/.octl extension, so my editor was called the Octl editor. The user interface was a C# application with some fancy touches (the artists love these) such as complete undo and redo system with states stacks and fancy looking buttons etc. It was a quick task between important projects, but still after it was done, some other programmer came for some discussions on how to integrate it with the in-house graphical manipulator (I did not get to see) which was rather unexpected but I was glad my program was built well enough for such integration to be also possible and simple.
About in the same period, I had a task to create a build pipeline, in effect to read some build descriptions and to generate a build script that in turn would get executed by the same program and ensure the build process keeps doing the reasonable things it was expected by starting the relevant processes and checking return codes. This system was later used for some tools and assets pipeline for a smaller project for Nintendo DS where the access to the resources was supposed to be implemented at quite a low level and the data was supposed to be loaded in a single chunk, modified with the actual memory addresses it was loaded at and be used as a fully functional set of structures in memory. These structures were containing assets such as sounds, letters for the fonts and their kerning characteristics and eventually my work carried me from having sounds and music as adpcm files and letters as png files on disk up to the point I had perfectly playing music and spot effects on the Nintendo DS screen and I also got to see the fonts displayed correctly and with the right colors per pixel :)

Also, I did work in the tools department at StainlessGames making tools also for Carmageddon: Reincarnation for almost a year; the tools I worked on were in more classes:
One class of tools were the PUPs: Small C# xaml applications designed to control various classes of attributes in the game engine. Sometimes these applications existed but were not compatible anymore with the codebase, other times they had to be implemented from scratch in order to control various new types of attributes. There were PUPs for lights for instance, to control the number of lights and the types of lights, a debug PUP to list debug information, then a PUP for the pools of shadows (implemented only by me), deciding what resolutions for the shadows classes allowing the artists to experiment with various shadow quality and numbers without harassing the 3d team, PUPs for camera orientation in game and various car related attributes and so on, all of them connecting to the game and eventually to the 3d manipulation program (Acolyte) including in some cases connecting as multiple instances, all of them serving the Lua virtual machine from the connected program with Lua code executed on the LuaVM thread and then with data distributed to the relevant classes and code called, followed in some cases by returning Lua code to be saved as persistent program state.

Another class of tools was Acolyte itself, which, based on the same extremely - experienced codebase as the game but never really being up to date to the latest engine code, was left on the shoulders of me and another member of the team for generic fixes and guest stars for their specific changes. Acolyte is a sort of 3dsMax in-house replacement, able to manipulate 3d objects and to edit game levels both for

Carmageddon and to a limited extent, Magic the Gathering – it is a very complex suite of plugins, in some cases even duplicating the functionality of the PUPs, always an apple of discontent when deciding the right development paths, MFC and C++ with almost the same Lua type control and a dedicated codebase in which every little bit of code might become explosively important especially sometimes later, but where not all the code is actually required (such as the SoundSystem itself is not, but the SoundSystem occlusion areas editor is needed), the development requests were lead by artists which were on extremely tight schedules and needed stuff implemented urgently and the QA was not able to help effectively until I've made some tools for them as well, to grab and compare screenshots from Acolyte vs the game itself and further modified the responsibilities cascade.
Also, I must mention my work on the 3ds Max plugin, which was – well – working, just needed some small fixes here and there to prevent it from crashing regularly on load and save and to update it to the latest 3ds Max version which obviously involved that older memory management functions could not be used, dependencies problems and in general a lot of configurations problems which dwarfed the actual fix by several orders of magnitude. This plugin was actually a family of related plugins with a common codebase with the entire engine, and when I took it as a task it did not even have continuous integration which ensured that every single commit broke the build and nobody knew before then; when I left Stainless both the import and export module were working with the latest code just a few months old, and the version for 3ds Max 2015 was compiling and working in a branch with – well – a vast merging.
And I did work a lot for various tools for KindGames.com, such as – say: tools for graphic people to edit collision areas in 2D characters and files related tools such as an assets pipeline where a puzzle file is created using C# from a set of png cutouts and jpg images and a text file to describe the original positions etc to be reinterpreted later in Flex ActionScript in the puzzle online, passing through various php scripts and in general, for every big project, separating as much of the code development into a tools section helps streamline the effort put into the actual game itself.

To speak a bit about code: to be honest, the best code there is is not written.
I am most proud of code I did not have to write either because I could reuse stuff that was already in a library, or perhaps code that could have been done without whilst still implementing the functionality.
As an example, in StainlessGames, there was an Acolyte plugin which was creating an object aligned to a surface's normal, and another which enabled an object to be grabbed and moved on X Y Z. It is obvious that the artists wanted to move the object always oriented to the surface, but as the design of the plugins was pretty stiff, this request was denied for a long time. When the task came to me, my solution was to not write any new code such as a third plugin to do the slide across surface solution but instead had one plugin use the other on the relevant condition; although not the most efficient solution, it was self-maintained, because there wouldn't be a dead code path to be used rarely, ensuring the functionality remains correct on unaware modifications and also pretty light in development time which was critical during that crunch to Alpha version.
Code that I have actually written, there was a combination of hot edited Lua code to be interpreted in a virtual Lua machine in the actual game engine itself and to be able to modify various program attributes and call functions run-time on multiple threads.
This was my procedurally generated clutter, in effect a system to generate various objects according to various factors, such as objects close to another (anchor) object and objects to appear on various surfaces; based on some flags in the Lua hot edited file, the created objects were oriented to the surface or not (such as trees always grow up, but grass can grow sideways), the density of the object was also given etc.

It was a very challenging part to actually get it to work. (Maybe it is not the best example, but I was so proud I printed a screenshot with all the source code open in all three languages (Lua, C++, C#) and placed somewhere visible. It was perhaps the most over-engineered bit of code I ever had to make work. Obviously, to actually generate some objects from an object hierarchy is not much, but it had to be done through these hoops, because the artists only had access to the Lua scripts and the GUI tools for them, and later the modding community actually only had the Lua scripts, so the (receiving) code also had to be perfect!

The technologies involved were Lua-related, Lua virtual machine-related, complicated stuff in C++, C# .xaml related run-time generated user interfaces, client-server, program states for undo – redo – store options etc.

So – the C# .xaml PUPs were connecting and then communicating Lua code back and forth with the C++ Acolyte or C++ Carmageddon over stream sockets, this Lua code was executed and interpreted in the Lua virtual machine, the functions were called, in some cases synchronization was needed to happen (such as when sending sound-related commands) and then the Lua code was sent back to be written to disk such as the state was persistent between sessions. The biggest problems were when something was not working; to debug and understand why (such as a too long delay in processing data, or connecting to the wrong socket, disconnecting and reconnecting to a new instance of one of the applications, some materials get broken on save or on undo on some machines), so to see this working perfectly made one feel proud.

Learning a new programming language - most of the now-days languages I program in are easy to learn because most of them are Object – Oriented and they sort of copy paradigms like ECMA script etc and they look a lot like C# and ActionScript. However, in the past, learning different programming languages meant learning per architecture ASM or Prolog. In learning Prolog the hardest bit was to completely understand the absolutely different mind-frame needed. It was all surreally recursive and (at the time) there was not much debug available so one either got it right or failed the Formal Logic exam. Another example was learning to program in the CLIPS inference engine, where until one gets it right, it seems easier to actually use real monkeys to activate the sprinklers.

Amongst the design-patterns I used in the past, as an example, I've used RAII, Object Pools for pools of shadow, Singleton for most anything unique including my take on a Sound System based on FMOD in Carmageddon and Factory for Sound Objects, Template method, Flyweight, Proxy again for Sound Objects that had their FMOD handles controlled in a single thread but their objects commands were accumulating in some synchronized lists, Command for sound commands such as modifications of pitch, volume and pan, Iterator, State, some of the Singletons were using Lazy Initialization.

To pick a very specific example of a design-pattern you have used for a specific case would be a case of Object Pools – I had some lights to manage in Acolyte for StainlessGames, and destroying the objects storing them was having some unwanted side-effects and the task at hand involved just the minimum dynamic lights use without fiddling into what happens on lights destruction and why.
Whether it the right choice, or in hindsight we could have chosen a better choice - right and wrong in programming choices are always biased by the relevant criteria.
Some design patterns are most always wrong, such as Singletons when they can be viewed as global variables in multi-threaded environments. There are always better ways, but everything is balanced by what is the desired outcome of the software.

Best regards,
Mihail Balescu
mihaiev@yahoo.com