Taking the Pole Position: Codemasters Leads the Pack in PC-to-Tablet Optimization with *GRID Autosport**

By Geoff Arnold

Games on mobile devices are undeniably proving both popular and lucrative. And given ever more powerful PCs, as well as the recently refreshed line up of consoles, things have never been better for high-end gamers craving graphics- and story-intensive experiences.

So where should a developer focus? For now, the best advice may be to aim at both ends of the PC-to-mobile spectrum, which are likely to coexist and even complement each other for years to come. That's certainly the opinion at Codemasters, the UK-based game developer behind the popular *GRID** series of racing games, including the refreshed *GRID Autosport*,* released in June 2014.

GRID Autosport, thanks to collaboration with Intel, takes full advantage of the 4th generation Intel® Core™ processors and runs well on mobile devices all the way down to 4-watt Intel® Atom™ processor-based tablets (among Intel's new multi-core system-on-chip [SoC] family formerly code-named Bay Trail).

"It's really important for us to cater to the whole market," said Richard Kettlewell, a senior programmer at Codemasters' headquarters in Southam, England. "We've made sure that the game has a set of comprehensive graphics options that can scale up to the top end, though still looks great if you're using a tablet."

Figure 1 shows a comparison of the game running on an Intel Atom processor-based tablet using the tablet's default graphics options and running on a 4th gen Intel Core processor-based tablet with its higher-quality defaults enabled.

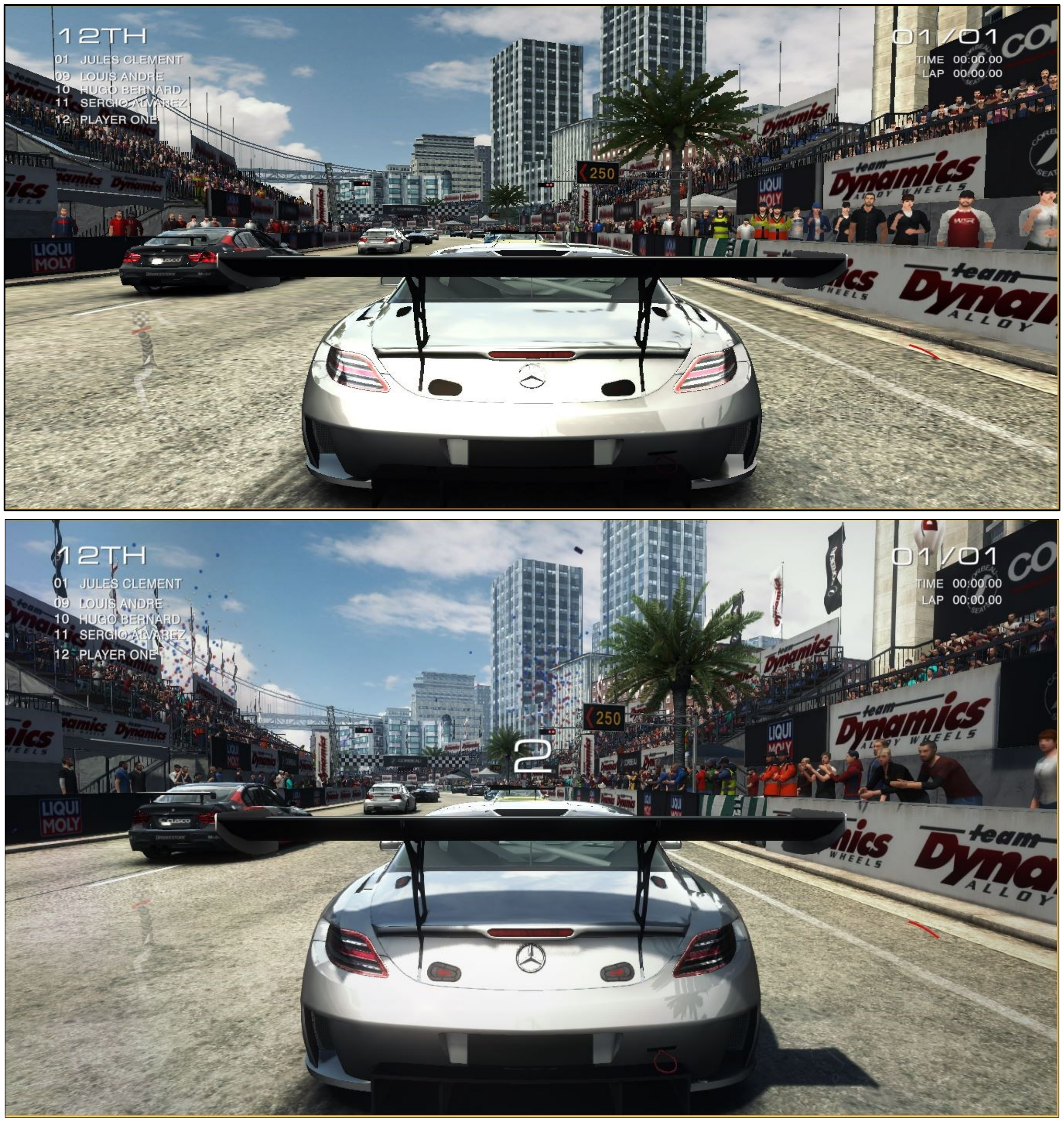


Figure 1: *Intel® Atom™ processor-based table using the game's tablet presets (top) and 4th generation Intel® Core™ processor-based table running the game with high-quality default settings (bottom).*

What follows is a description of how Kettlewell and producer Toby Evan-Jones, Kettlewell's colleague, achieved their goal by applying new techniques and algorithms enabled by 4th gen Intel Core processors (programmable blending, adaptive order independent transparency (AOIT), and adaptive volumetric shadow maps [AVSM]) and scaling down to ultra-low-voltage tablets in large part by just adjusting shaders responsible for high-frequency details. Codemasters also successfully juggled multiple DirectX* swap chains to implement a dual-screen option, one that shows the race from a different perspective and that further extends options for tablet gaming both in the living room and online.

From the Early Days of Multi-Core to the Explosion of Mobile

Codemasters' ongoing effort to make use of the better graphics technology in the processors, the improved CPU performance, and Intel® Iris™ graphics extensions to the DirectX API paid off. Last year, GRID 2 earned the top spot on the UK Video Game Chart, and reviewers implicitly noted the benefits of applying two algorithms, including enhanced smoke-particle shadows and lighting as well as improved foliage transparency. The AOIT and AVSM algorithms became practical for in-game deployment only with the arrival of 4th gen Intel Core processors.

Intel's relationship with Codemasters goes back several years. In the mid-2000s, to support the move to dual- and quad-core processors, Intel provided engineering support to Codemasters to help with multi-threaded coding and testing, part of a broader effort to enable software developers around the world to take advantage of the new chip architectures. Next was Intel's extensive work with Codemasters on graphics optimizations in *GRID 2*, released in May 2013. The results are described in a June 2013 case study and in more technical detail in a presentation by Kettlewell and Intel application engineer Leigh Davies delivered at GDC 2014 (Figure 2). (Kettlewell's 2014 GDC video: https://software.intel.com/en-us/videos/intel-and-codemasters-advanced-rendering-techniques)

Figure 2: *Richard Kettlewell at GDC 2014*

What's New in *GRID Autosport*: Programmable Blending and More

GRID Autosport incorporates and extends much of the earlier learning about AOIT and AVSM. In addition to the goal of simultaneously targeting low-power tablets, there have been at least three other development firsts in the release of *GRID Autosport*.

First, the team decided to focus on the PC as the lead platform. Making sure the game looked and behaved well on PCs was a first order of business and a common-sense reaction to the fact that PC performance now far exceeds that of Codemasters' targeted console platforms. And when it comes to gameplay, PCs are surging in popularity. Codemasters' internal data reflects this trend. "When we were looking at the sales for *GRID 2*, we noticed a marked jump in PC sales in comparison to consoles," said Evan-Jones.

The PC's popularity is also reflected in what's happening with online gameplay based on data from Codemasters' servers. In the first quarter of 2012, just 16 percent of online gamers accessed Codemasters servers via Steam* (generally used only by PC gamers). A year later, Steam gamers accounted for 38 percent of all online gameplay, a trend that has continued.

Second, in addition to AOIT and AVSM, the Codemasters team for the first time applied programmable blending, enabled by the Pixel Shader Ordering graphics extension available on 4th gen Intel Core processors. The extension allows for a fine degree of control over blending colors and brightness.

Third, making use of Intel® Wireless Display (Intel® WiDi), Codemasters created the option to view the game action on a second monitor. The primary screen shows the player's view, while the second screen shows the race much as it would be viewed by spectators at the event, including supplemental information such as the leaderboard. This second image makes sense as far as potentially streaming gameplay, an exploding market most exemplified by Twitch, much in the news lately because of reports of its pending sale to YouTube for USD 1 billion. More on how Codemasters implemented this second-screen option later.

Brighter Lights, Bigger High-Dynamic Range

New developments aside, the main goal in updating *GRID Autosport* was an old one: engaging and thrilling the person behind the virtual steering wheel. Achieving this goal meant focusing on realism in everything, including handling and traction, the AI-powered opponents, and the scenery that flies by as drivers approach a hairpin turn at 140 miles per hour (mph).

Codemasters helps deliver this realism with its application of Intel's PixelSync, a DirectX extension enabling a Read/Modify/Write on certain texture surfaces in games. This approach enables programmable blending to allow for some interesting effects, especially when it comes to sunlight or floodlights seen through the car's windshield. In *GRID 2*, this sort of light was much dimmer than the artists would have preferred. "Programmable blending allows us to get the brightness back into those objects without darkening down the rest of the scene," said Kettlewell.

In their GDC presentation, Kettlewell and Davies showed images of the sun coming through the windshield before and after programmable blending was applied (Figure 3). As they describe it, because the high dynamic range (HDR) lighting values were encoded logarithmically into an R10G10B10A2 back buffer, the fixed-function alpha blending of encoded values was invalid. The result was a loss of HDR behind transparencies. The solution: apply programmable blending to blend in linear space.

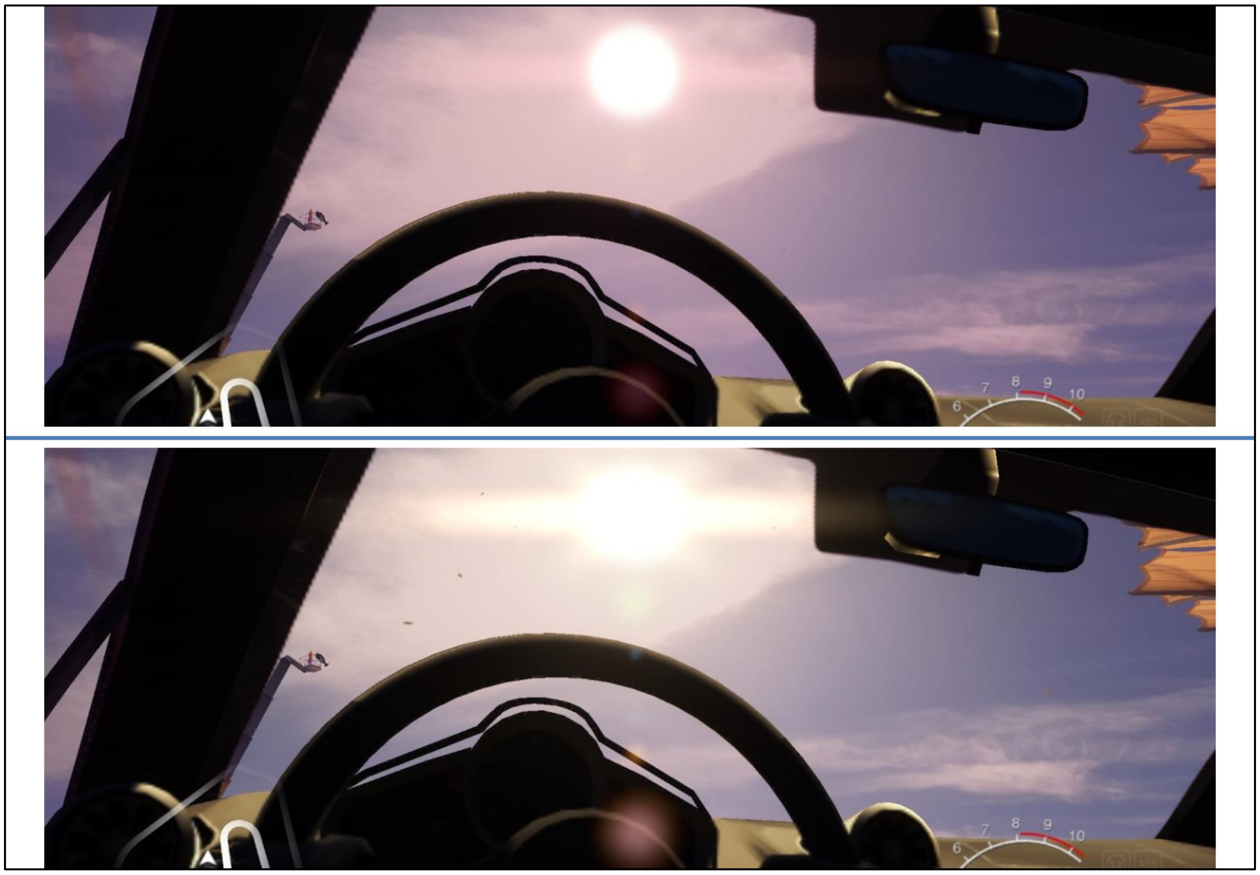


Figure 3: *Screen shots of sunlight through the windshield before (top) and after (bottom) programmable blending is applied. Courtesy Kettlewell/Davies 2014 GDC presentation (see slides 27 and 28)*

More about AOIT and AVSM

As with *GRID 2*, Codemasters made much use of AOIT and AVSM. AOIT improved the rendering of foliage and semi-transparent trackside objects such as fences. Like programmable blending, the algorithm also makes use of Intel's PixelSync, which provides ordered Read/Modify/Write for a given pixel.

"If two pixels in flight are being rendered to the same screen location at the point of the synchronization primitive in the pixel shader, only one shader is allowed to continue, and the one chosen is dependent on the order submitted to the front end," wrote Davies in a July 2013 blog post that includes an OIT sample implementation. "The remaining shader(s) resume once the first shader has completed in the order they were submitted." The central AOIT code is shown in Figure 4. (Davies notes as a caveat that the code doesn't include any of the setup or resolving of data, or explanation of the UAV textures that are referenced. The best reference, he said, is to see the sample in the 2013 blog post.)

```
void AOITSPInsertFragment(in float fragmentDepth,
                          in float  fragmentTrans,
                          in float3 fragmentColor,
                          inout ATSPNode nodeArray[AOIT_NODE_COUNT])
```

```
{
      int i, j;

      float  depth[AOIT_NODE_COUNT + 1];
      float  trans[AOIT_NODE_COUNT + 1];
    uint color[AOIT_NODE_COUNT + 1];

      ///////////////////////////////////////////////////
      // Unpack AOIT data
      ///////////////////////////////////////////////////
    [unroll] for (i = 0; i < AOIT_NODE_COUNT; ++i) {
      depth[i] = nodeArray[i].depth;
      trans[i] = nodeArray[i].trans;
      color[i] = nodeArray[i].color;
      }

      // Find insertion index
    int index = 0;
      float prevTrans = 255;
      [unroll] for (i = 0; i < AOIT_NODE_COUNT; ++i) {
      if (fragmentDepth > depth[i]) {
            index++;
            prevTrans = trans[i];
      }
      }

      // Make room for the new fragment. Also composite new fragment with the
current curve
    // (except for the node that represents the new fragment)
      [unroll]for (i = AOIT_NODE_COUNT - 1; i >= 0; --i) {
      [flatten]if (index <= i) {
            depth[i + 1] = depth[i];
            trans[i + 1] = trans[i] * fragmentTrans;
            color[i + 1] = color[i];
      }
      }

                  // Insert new fragment
                  const float newFragTrans = fragmentTrans * prevTrans;
                  const uint  newFragColor = PackRGB(fragmentColor * (1 -
fragmentTrans));
                  [unroll]for (i = 0; i <= AOIT_NODE_COUNT; ++i) {
                                [flatten]if (index == i) {
                                                depth[i] = fragmentDepth;
                                                trans[i] = newFragTrans;
                                                color[i] = newFragColor;
                                }
                  }

      // pack representation if we have too many nodes
            [flatten]if (depth[AOIT_NODE_COUNT] != AOIT_EMPTY_NODE_DEPTH) {
```

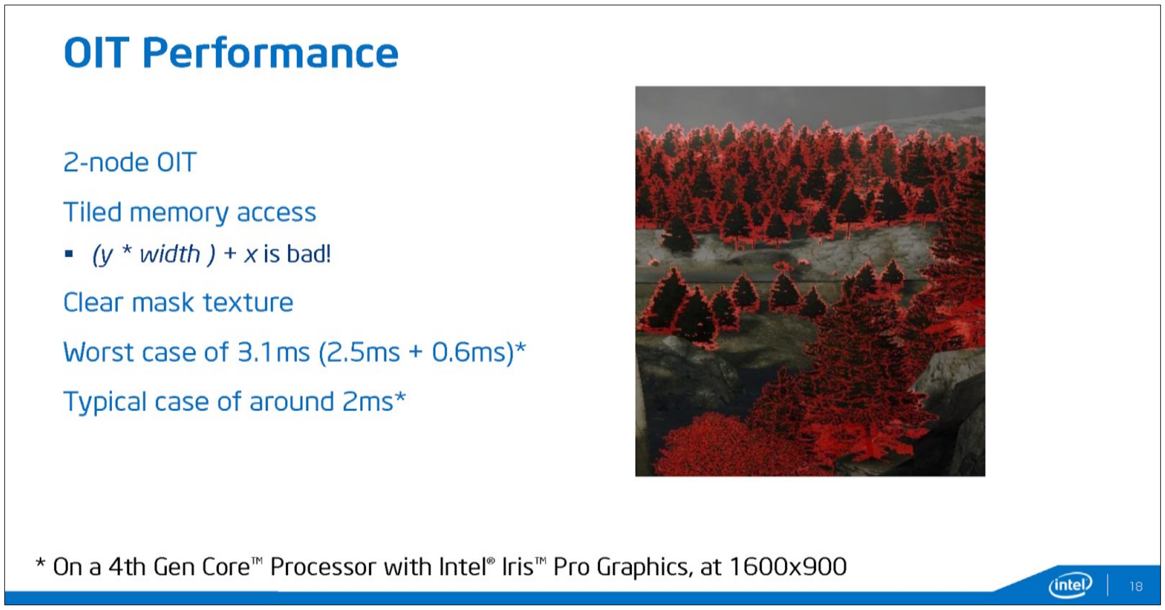
```
                                float3 toBeRemovedCol =
UnpackRGB(color[AOIT_NODE_COUNT]);
                                float3 toBeAccumulCol =
UnpackRGB(color[AOIT_NODE_COUNT - 1]);
                                color[AOIT_NODE_COUNT - 1] =
PackRGB(toBeAccumulCol + toBeRemovedCol * trans[AOIT_NODE_COUNT - 1] *

rcp(trans[AOIT_NODE_COUNT - 2]));
                                trans[AOIT_NODE_COUNT - 1] =
trans[AOIT_NODE_COUNT];
                  }

    // Pack AOIT data
      [unroll] for (i = 0; i < AOIT_NODE_COUNT; ++i) {
      nodeArray[i].depth = depth[i];
      nodeArray[i].trans = trans[i];
      nodeArray[i].color = color[i];
      }
}
```

Figure 4: *Central AOIT code*

At GDC, Kettlewell and Davies described several optimizations to the OIT code (Figure 5), including tiled memory access to improve memory coherency and overall performance, as well as the use of a mask texture to conserve bandwidth.



OIT Performance

2-node OIT

Tiled memory access

- *(y * width)* + *x* is bad!

Clear mask texture

Worst case of 3.1ms (2.5ms + 0.6ms)*

Typical case of around 2ms*

* On a 4th Gen Core™ Processor with Intel® Iris™ Pro Graphics, at 1600x900

Figure 5: *Using tiled memory access in AOIT and AVSM to improve memory coherency. Courtesy Kettlewell/Davies 2014 GDC* presentation *(see slide 17)***.**

Originally, Codemasters was reading and writing to the OIT data structures in a linear format. Switching to a tiled access pattern saved half a millisecond per frame, which was about two percent of total frame time. The idea of using tiled memory came from the common practice of swizzling texture formats to optimize their memory access patterns. The clear mask is a texture used to flag the pixels that contain OIT data. Using a clear mask saves a lot of bandwidth when figuring out if there is any OIT data at a given location. The alternative is to read the OIT data and check if it has been initialized, which requires reading a much larger structure.

The code snippet in Figure 6 describes how to implement the tiled addressing in AOIT:

```
uint AOITAddrGenUAV(uint2 addr2D)
{
            uint2 dim;
                  gAOITSPClearMaskUAV.GetDimensions(dim[0], dim[1]);
                  return AOITAddrGen(addr2D, dim[0]);
}

uint AOITAddrGen(uint2 addr2D, uint surfaceWidth)
{
#ifdef AOIT_TILED_ADDRESSING

                  surfaceWidth        = surfaceWidth >> 1U;
                  uint2 tileAddr2D  = addr2D >> 1U;
                  uint  tileAddr1D  = (tileAddr2D[0] + surfaceWidth *
tileAddr2D[1]) << 2U;
                  uint2 pixelAddr2D = addr2D & 0x1U;
                  uint  pixelAddr1D = (pixelAddr2D[1] << 1U) +
pixelAddr2D[0];

                  return tileAddr1D | pixelAddr1D;
#else
                  return addr2D[0] + surfaceWidth * addr2D[1];
#endif
}
```

Figure 6: *Code implementing tiled memory addressing in AOIT*

AVSM was used to simulate light traveling through transparent objects. Its application is particularly noticeable when cars kick up plumes of smoke and dust. AVSM works much like AOIT, but instead of the nodes storing color information, each pixel of the shadow map stores a compact approximation to the transmittance curve along the corresponding light ray, which helps make these plumes more realistic. The code for deciding which nodes to remove becomes more complex than AOIT; the goal is to minimize the change in area under a graph charting the transmittance, similar to the one in Figure 7 taken from the GDC presentation.

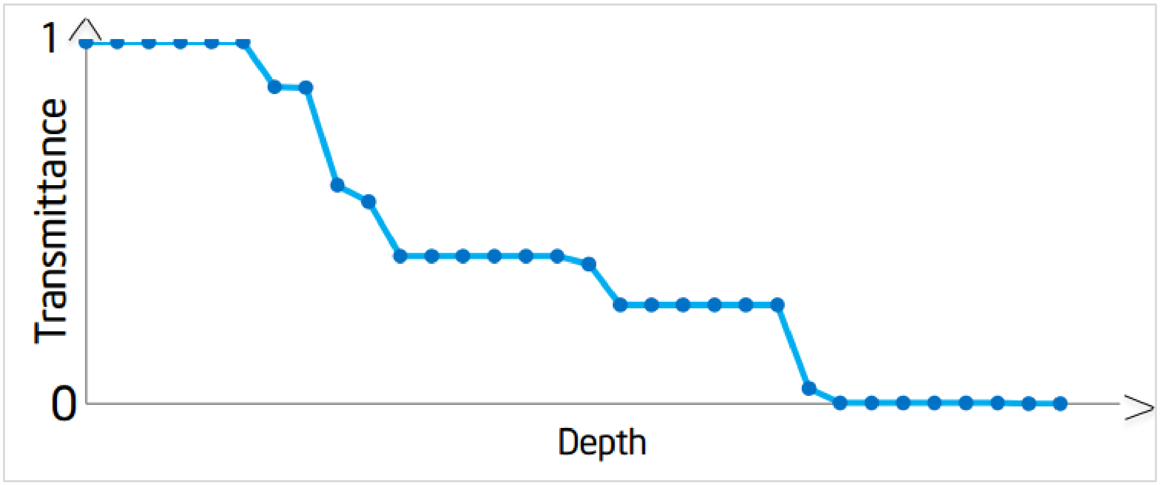


Figure 7: *Transmittance graph versus depth from a light source*

Figure 8 shows an example of the code used to calculate the node for removal.

```
void AVSMGenInsertFragment(in float fragmentDepth,
in float  fragmentTrans,
inout AVSMGenNode nodeArray[AVSM_NODE_COUNT])
{
    int i, j;
    float  depth[AVSM_NODE_COUNT_CUT + 1];
    float  trans[AVSM_NODE_COUNT_CUT + 1];

    ///////////////////////////////////////////////////
    // Unpack AVSM data removed for simplicity

    // Find insertion index
    int index = 0;
    float prevTrans = 1.0f;
    [unroll] for (i = 0; i < AVSM_NODE_COUNT_CUT; ++i) {
        if (fragmentDepth > depth[i]) {
            index++;
            prevTrans = trans[i];
        }
    }

    // Make room for the new fragment. Also composite new fragment with the
current curve
    // (except for the node that represents the new fragment)
    [unroll]for (i = AVSM_NODE_COUNT_CUT - 1; i >= 0; --i) {
        [flatten]if (index <= i) {
            depth[i + 1] = depth[i];
            trans[i + 1] = trans[i] * fragmentTrans;
        }
    }

    // Insert new fragment
    [unroll]for (i = 0; i <= AVSM_NODE_COUNT_CUT; ++i) {
        [flatten]if (index == i) {
            depth[i] = fragmentDepth;
            trans[i] = fragmentTrans * prevTrans;
        }
    }

    // pack representation if we have too many nodes
    [branch]if (depth[AVSM_NODE_COUNT_CUT] != AVSM_GEN_EMPTY_NODE_DEPTH) {
        // That's total number of nodes that can be possibly removed
        const int removalCandidateCount = (AVSM_NODE_COUNT_CUT + 1) - 1;
       const int startRemovalIdx = 1;

        float nodeUnderError[removalCandidateCount];
        [unroll]for (i = startRemovalIdx; i < removalCandidateCount; ++i) {
            nodeUnderError[i] = (depth[i] - depth[i - 1]) * (trans[i - 1] -
trans[i]);
        }
        // Find the node the generates the smallest removal error
        int smallestErrorIdx = startRemovalIdx;
        float smallestError  = nodeUnderError[smallestErrorIdx];

        [unroll]for (i = startRemovalIdx + 1; i < removalCandidateCount; ++i)
{
            [flatten]if (nodeUnderError[i] < smallestError) {
                smallestError = nodeUnderError[i];
                smallestErrorIdx = i;
            }
        }

        // Remove that node..
        [unroll]for (i = startRemovalIdx; i < AVSM_NODE_COUNT_CUT; ++i) {
            [flatten]if (smallestErrorIdx <= i) {
                depth[i] = depth[i + 1];
            }
        }
        [unroll]for (i = startRemovalIdx - 1; i < AVSM_NODE_COUNT_CUT; ++i) {
            [flatten]if (smallestErrorIdx - 1 <= i) {
                trans[i] = trans[i + 1];
            }
        }
    }

    ///////////////////////////////////////////////////
    // pack AVSM data removed for simplicity
}
```

Figure 8: *AVSM calculation of the smallest error metric*

Changing Gears: Implementing a Tablet Version

The optimizations in *GRID Autosport* for 4th gen Intel Core processors are only part of the story. The Codemasters team built the game so that it will deliver a satisfying experience on a tablet with an ultra-low voltage Intel Atom processor. The fact that the same executable file can span the 4–300-watt range is astonishing, a bit like designing a racetrack in the real world that's equally fun for go-karts that chug along at 20 mph and Formula 1 cars that speed around at more than 200 mph.

The process of creating a tablet-ready version proved somewhat counterintuitive, or at least surprising. The team first took the traditional approach to reducing the game's minimum hardware requirement, making a list of graphical elements they assumed would have to be cut outright because of their processing overhead and then lowering other scalable elements in the game (such as texture resolution for shadow and environment maps). What quickly became apparent was that the removal of some effects, such as the complex depth of field and motion blur in the final post-processing, meant many of the current pixel shaders were doing redundant work that could be stripped out.

Rather than modifying the existing shaders, the team took a different approach: find the simplest shaders in the game used for creating the reflection maps on the normal settings and apply those to the main scene render, enabling only the effects that could be achieved with the available data. At the end of this pass, to their surprise, a very playable version of the game hummed along at 40-45 frames per second, a fast enough frame rate to show that they had cut too much.

Next was the unanticipated task of deciding which visual elements to add back in by trying to weigh the gameplay benefits of certain visual effects against the processing overhead. Yes, sometimes entire scene elements were swapped out. 3D models of crowds and trees gave way to more 2D billboards lining the track, though many of the graphical elements on the original cut list were re-enabled. However, the fidelity of existing elements in the high-end version was oftentimes simply dialed down. For example, for those playing on PCs, the dynamic reflections of buildings that appear briefly in a car's paintwork as it races down city streets were replaced by a static environment map. Also simplified were real-time shadows of nearly anything except for the cars in the race.

One of the team's takeaways is that, when designing for tablets, thinking first about all the objects that might need to be removed from the game is not always best. Codemasters left in a majority of the most central models and merely adjusted the shaders responsible for rendering the surface properties to remove high-frequency details. A good example was the use of specular reflection maps on many surfaces in the game. On high-resolution desktop displays, the effects were noticeable on screen. However, on a 10-inch tablet screen they were almost imperceptible on many objects, the exception being the car wheels. Ultimately, these effects were enabled for only wheels and objects that truly benefited. From the perspective of gamers who are used to playing on a high-end PC, still the primary audience, less vivid rendering of a road's surface or building's windows might not be noticeable on a tablet's relatively small screen. However, elements that go missing—say, a person who is seen standing next to a car in the PC version and who isn’t there in the tablet version—might be conspicuous.

Another conclusion, true of so much software, is that work done now often yields additional benefits. The improvements to the shaders used for the tablet version yielded both visual gains and performance improvements for the reflection maps that also used the same shaders for the higher-end systems. In addition, there is the basic feature of computing technology, which tends to get more powerful and less expensive over time. Efforts to simply maintain and update even a mid-range version of the game might have a later payoff, because, in terms of performance, today's mid-range PC is tomorrow's tablet or phone. The moral of the story is to never stop optimizing.

Second Screen, with a Twist

Understanding that hardware capabilities and consumer tastes change over time, Codemasters used Miracast* to add a dual-screen mode (Figure 9) where the PC screen is the traditional racing car point-of-view and the second screen shows a spectator point of view, similar to what you might see while watching a Formula 1 race on TV.

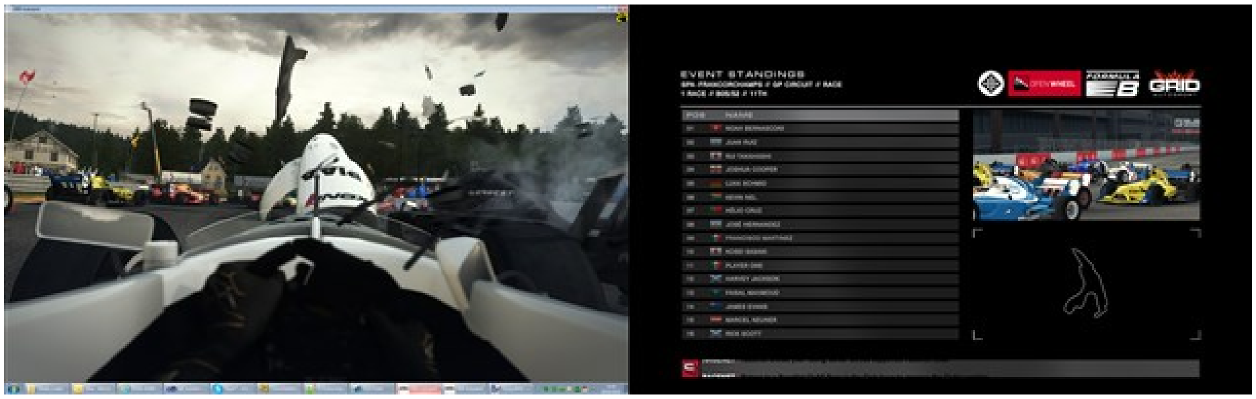


Figure 9: *Players can connect via Miracast* to a second screen (TV) to show a spectator point-of-view that includes additional information, such as event standings*

Although this feature is still experimental for *GRID Autosport*, offering it is a great way for Codemasters to see how people use it and to extend a scenario that has become commonplace: connecting a mobile device to a TV and sharing the same image with the bigger screen.

While optimizing for high-end graphics consumed most of their time, the Codemasters developers still ran into a few roadblocks in trying to implement the dual-screen feature. The main one, said Kettlewell, was creating multiple DirectX swap chains, the collection of buffers used for displaying frames to a user. “When creating such swap chains, it is best to create all swap chains as windowed, and then set them to full screen,” according to MSDN. The approach added complexity in managing the user experience. Codemasters wanted to give users the option of turning on the second display within the game while not forcing the second display because some people might prefer to have another application visible.

.Another issue that caused problems at the start was that the game’s engine stored some global states for the window that were modified in the callback function for the windows loop. Adding an extra window meant that two very different displays could call this function, creating confusion over the global state—always a challenging bug to track down.

“Multiple swap chains is not something that I think is widely supported,” said Kettlewell. “It caused problems with certain bits of software we use. We wound up in conversations with vendors about updating drivers. Overall it’s been quite a challenge, but that just goes to show how new this idea is of two screens, each showing a unique image.”

Adopting new technology such as Miracast is paying off in unexpected ways. Within days of the game's launch, users were publishing images, such as the one in Figure 10, on the Codemasters forums.

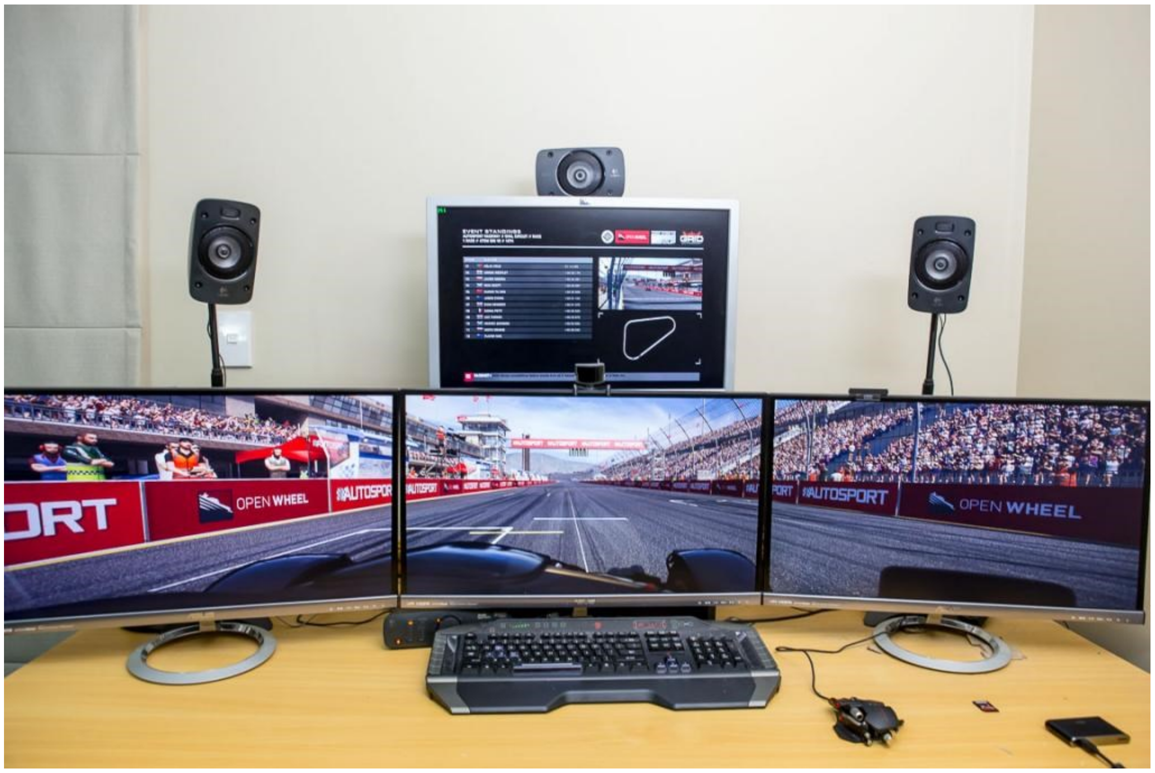


Figure 10: *The second screen experience used as part of a more traditional multi-monitor setup over an HDMI* cable*

The positive feedback on the second screen feature shows that technology primarily designed for new usage models, such as PC gaming on a tablet in the living room, can still benefit other players when implemented with care as a genuine expansion of the gamer's experience. Indeed, Codemasters' work to enable various mobile use cases was so new that the ability to control the game using the tablet's touch screen didn't get into the release. Previous *GRID* titles didn't support mouse input, so the legacy codebase didn't have any concept of which element was being clicked or tapped based on only the coordinates of the image on the screen. However, touch navigation will be patched into the game soon (Figure 11). Davies has already shot video of the team testing the touch navigation in the office. "It's definitely the last piece of the puzzle in terms of fully stretching *GRID Autosport* so it's accessible on a tablet," said Evan-Jones. (The video Davies shot is available here: https://www.dropbox.com/s/bymr865ovzfcs3c/WP_20140529_020.mp4 or here: https://www.dropbox.com/s/sci3j1f70ziegov/Bay%20Trail%20.MOV)

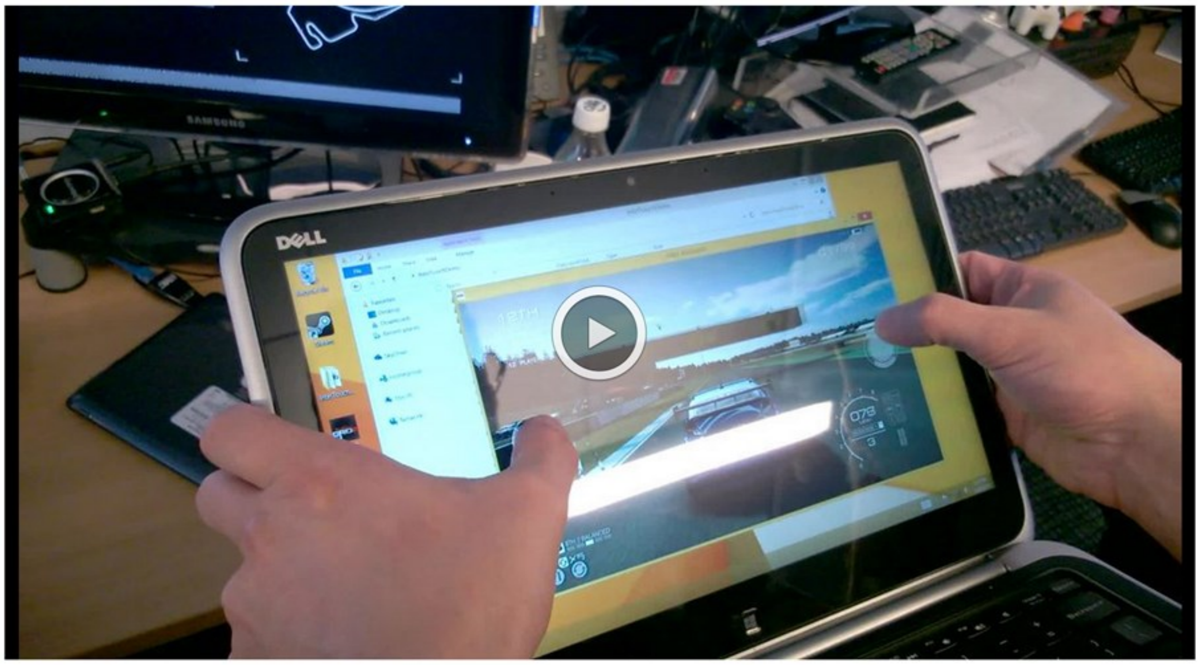


Figure 11: *Raw video shot by Davies at Codemasters' offices showing touch navigation, a feature that will be patched into the game soon*

In addition to completing the tablet experience, touch opens up new play modes on more traditional touch-enabled laptops and Ultrabook™ devices, allowing for the same game to be played anywhere with a choice of input controls.

Just Do...Both

So do tablets and phones represent the kind of disruptive innovation that the tech industry is known for? Maybe or maybe not. More certain is that even though most people aren't ditching their PCs outright for smaller portable devices, ultra-low-voltage portables seem likely to remain among the hottest categories in tech for the foreseeable future.

"As exciting as it is to focus on high-end PCs, if you focus just on the high end, you're really looking at a niche corner of the market rather than a much wider audience," said Kettlewell. In other words, if the question is, should you develop your game for the screaming next-gen PC or the humble tablet, there is only one answer: do both.

Additional Resources

GRID Autosport: http://www.codemasters.com/uk/gridautosport/pc/

Codemasters GRID 2 on 4th Generation Intel® Core™ Processors - Game Development Case Study*: https://software.intel.com/en-us/articles/codemasters-grid-2-on-4th-generation-intel-core-processors-game-development-case-study

Rendering in Codemasters' GRID2 and beyond: Achieving the ultimate graphics on both PC and tablet: https://software.intel.com/sites/landingpage/gdc2014/pdfs/thu-rendering-in-codemasters-grid2-and-beyond-v10.pdf

Developer's Guide for Intel® Processor Graphics (For 4th Generation Intel® Core™ Processors): https://software.intel.com/sites/default/files/4th-gen-core-graphics-dev-guide.pdf

Programmable Blend with Pixel Shader Ordering: https://software.intel.com/en-us/blogs/2013/03/27/programmable-blend-with-pixel-shader-ordering

Share Your Screen With Intel® Wireless Display (Intel® WiDi): http://www.intel.com/content/www/us/en/architecture-and-technology/intel-wireless-display.html

Order-Independent Transparency Approximation with Pixel Synchronization: https://software.intel.com/en-us/blogs/2013/07/18/order-independent-transparency-approximation-with-pixel-synchronization

Adaptive Volumetric Shadow Maps: https://software.intel.com/en-us/blogs/2013/03/27/adaptive-volumetric-shadow-maps/

Notices